(4)

UNISYS

INTEGRATING SYNTAX, SEMANTICS, AND DISCOURSE
DARPA NATURAL LANGUAGE UNDERSTANDING PROGRAM

R&D FINAL REPORT
Unisys Defense Systems
Contract Number: N00014-85-C-0012

Volume II -- DOCUMENTATION

ARPA ORDER NUMBER: 5262
PROGRAM CODE NO. NR 049-602 dated 10 August 1984 (433)
CONTRACTOR: Unisys/Defense Systems
CONTRACT AMOUNT: 1,704,901
CONTRACT NO: N00014-85-C-0012
EFFECTIVE DATE OF CONTRACT: 4/29/85
EXPIRATION DATE OF CONTRACT: 9/30/89
PRINCIPAL INVESTIGATOR: Dr. Lynette Hirschman PHONE NO. (215) 648-7554

SHORT TITLE OF WORK: DARPA Natural Language Understanding Program

REPORTING PERIOD: 4/29/85-9/30/89

TABLE OF CONTENTS - VOL. II DOCUMENTATION

A. PUNDIT USER'S GUIDE , VERSION 1.2

B. PUNDIT LEXICAL ENTRY PROCEDURE USER's GUIDE , VERSION 1.1

C. PUNDIT LEXICAL ENTRY PROCEDURE MINI-RELEASE NOTES ,VERSION 1.2

D. A GUIDE TO OBJECT OPTIONS IN PUNDIT

E. A USER'S GUIDE TO THE SELECTION MODULE

F. PUNDIT'S SYNTACTIC COMPONENT: DESCRIPTION OF COVERAGE

G. LEXICAL LOOK-UP PROCEDURE IN PUNDIT

H. PROLOG STRUCTURE EDITOR

I. SYSTEM ADMINISTRATION FOR PUNDIT (SAP)

PUNDIT
User's Guide*

Version 1.2

July 6, 1988

Unisys Logic-Based Systems
Paoli Research Center
P.O. Box 517, Paoli, PA 19301

*This work has been supported by DARPA contract N00014-85-C-0012, administered by the Office of Naval Research.

Contents

1 Introduction 1

1.1 The *User's Guide* . . . 1

1.2 The Software . . . 1

2 Running PUNDIT 2

2.1 Core Images and Domain Images . . . 2

2.2 The MUCK Domain . . . 2

2.3 **parse** and **pundit** . . . 2

2.4 Before You Begin . . . 3

2.5 Processing a Sentence . . . 3

3 Interpreting PUNDIT Output 5

3.1 The Parse Tree . . . 5

3.2 The ISR . . . 5

3.3 The IDR . . . 9

4 Commonly Used Procedures 12

4.1 edit_rule . . . 12

4.2 edit_word . . . 12

4.3 parse . . . 12

4.4 pundit . . . 13

4.5 punt . . . 15

4.6 rdb_remove . . . 15

4.7 readIn . . . 16

4.8 squery . . . 16

4.9 ssucceed . . . 16

4.10 switches . . . 17

4.10.1 enter_new_word . . . 18

4.10.2 np_trace . . . 18

4.10.3 parse_tree . . . 18

4.10.4 conjunction 19
4.10.5 semantics 19
4.10.6 translated_grammar_present 19
4.10.7 translated_grammar_in_use 19
4.10.8 grinder 19
4.10.9 text_mode 20
4.10.10 decomposition_trace 20
4.10.11 summary 20
4.10.12 show_isr 21
4.10.13 selection 21
4.10.14 enable_db_access 21
4.10.15 count 21
4.10.16 all_time 21
4.10.17 time_trace 22
4.10.18 window_display 22

A Installing the System **23**

B Building PUNDIT Images **24**

B.1 Building a Core PUNDIT Image 24
B.2 Creating a Functional Core PUNDIT Image 24
B.3 Creating a Complete Domain-Specific Image 25

C Customizing Your PUNDIT User Environment **26**

D PUNDIT Files and Dependencies **27**

D.1 Files 27
D.2 Dependencies 30

E PUNDIT Bibliography **32**

E.1 Background Reading 32
E.2 Papers and Presentations 32
E.3 Technical Documentation 34

List of Figures

1 Running PUNDIT 4

2 A glossary of string-grammar terms 6

3 Parse tree for *Visual sighting of periscope followed by attack with asroc and torpedos.* 7

4 ISR for *Visual sighting of periscope followed by attack with asroc and torpedos.* 8

5 .DR for *Visual sighting of periscope followed by attack with asroc and torpedos.* 11

6 Using the **pundit** procedure 14

7 Using the **rdb_remove** utility 15

8 Using the **switches** utility 17

9 Setting the **grinder** switch 20

10 Sample **prolog.ini** file 26

1 Introduction

1.1 The *User's Guide*

The *PUNDIT User's Guide* is intended to provide a concise and general introduction to the facilities of the PUNDIT text-processing system. The intended audience is computational linguists familiar with Quintus Prolog. While this document is not a reference manual, and does not in itself contain sufficient information for you to either extend the system or port it to a new domain, we have tried to cover the operational basics: how to run PUNDIT (Section 2) and how to interpret PUNDIT's output (Section 3). In addition, Section 4 documents the two main procedures for accessing the system (`parse` and `pundit`), as wel' as a number of other procedures which we make frequent use of as developers. Appendix A and Appendix B will help you set the system up. Appendix D identifies the core and domain files, and Appendix E lists papers, presentations, and technical documentation available for PUNDIT.

1.2 The Software

The *User's Guide* is designed to accompany a subset of the text-understanding software which has been developed at the Paoli Research Center, as it exists on the date of publication: the core components of PUNDIT, together with the domain-specific components developed to process Navy tactical messages (RAINFORMs). This domain will be referred to henceforth as the MUCK domain (an acronym for the message understanding conference which occasioned the development of the software). The MUCK software is essentially similar to that developed for other domains, and may be considered representative: it includes a domain-specific message input screen, lexicon, knowledge base, semantics rules and database definitions, and it supports both analysis of text and limited natural language queries. It differs from other domain software chiefly in having a comparatively rich knowledge base.

2 Running PUNDIT

2.1 Core Images and Domain Images

Before you can use PUNDIT, the software must be installed at your site and the images built. Appendix A contains instructions for creating a PUNDIT core image and a MUCK domain image.

The core image is not functional, and is generally used only to build the domain images.[1] In the discussion that follows, it will be assumed that you have a MUCK domain image available to you.

2.2 The MUCK Domain

The MUCK domain has been designed to process the *Remarks* field of Navy tactical messages. Since the formatted fields in these messages contain information which establishes the initial context for interpreting the text (message originator, date/time, etc.), we have developed a special front-end to collect this information. This message front-end is accessed by issuing the command `pundit`. See Section 4 for more information about this command.

In order to make use of the MUCK domain image for syntactic and semantic analysis of natural language input, you will need to know something about the sublanguage and the knowledge base for this domain. In the file `muck_working.pl` you will find a subset of the messages from our message corpus which PUNDIT is currently able to process. By examining other domain-specific files such as the lexicon, the knowledge base, and the semantics rules, you should be in a position to construct your own input (see Appendix D for a list of these files).

2.3 parse and pundit

The `pundit` command (discussed above) invokes the domain-specific message processing front-end to the system, which collects both message header information and the message body. An alternative, domain-independent method of accessing the system is provided by **parse**, which prompts only for the text to be processed. Many of the researchers working on PUNDIT currently interact with the system using **parse**, although certain higher-level processes—reference resolution in particular—do not perform as well as they otherwise could, since the initial discourse context is empty. The **parse** command, however, provides more options for developers, and is the only command to use when no semantic processing is desired (the front-end invoked by `pundit` assumes that a complete analysis is required). These two commands are discussed in more detail in Section 4.

[1] The core image contains only the core procedures of PUNDIT, including the core lexicon (see Appendix D). See Appendix B for details on how to create a functional image from the core image.

2.4 Before You Begin

Since we will be using a text from the MUCK domain to illustrate PUNDIT's operation, at this point you may wish to load the MUCK image. Before using **parse** or **pundit**, however, you will first need to set a few of the software switches which enable or disable various system features. Do this by executing the **switches** procedure (described in more detail in Section 4). The **switches** procedure will display the current switch settings in the image, and will prompt you for a list of switches to be changed. Make sure, at least for now, that you have the following switches turned on, and that all the others are turned off:

1. **parse_tree**
2. **conjunction**
3. **semantics**
4. **translated_grammar_present**
5. **translated_grammar_in_use**
6. **selection**

At this stage you may also want to tell the Selection module not to query you about new co-occurrence patterns. Call the procedure **ssucceed** (see Section 4 for more details).

2.5 Processing a Sentence

Having brought up the MUCK domain image and set your switches, you are are now ready to analyze a sentence. Call **parse**, and you should see the prompt "**sentence:**". Since the following section describes the output generated from processing the sentence *visual sighting of periscope followed by attack with asroc and torpedos.*, you might want to type it in now, including the final period. After typing the sentence in, you will need to signal the end of input by entering **two carriage-returns**. The following is a transcript of someone doing what you have just been asked to do in the last two subsections[2].

[2]Note that if you later create a **prolog.ini** file, as described in Appendix C, your initial switch settings may differ from those shown in the figure.

```
%/nlp/nlp/pundit/muck/Muck.qimage

Quintus Prolog Release 2.2 (Sun-3, Unix 3.2)
Copyright (C) 1987, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- switches.

1. enter_new_word------------------> OFF
2. np_trace------------------------> OFF
3. parse_tree----------------------> OFF
4. conjunction---------------------------------> ON
5. semantics-----------------------> OFF
6. translated_grammar_present------------------> ON
7. translated_grammar_in_use-------> OFF
8. grinder-------------------------> OFF
9. text_mode-----------------------> OFF
10. decomposition_trace------------> OFF
11. summary------------------------> OFF
12. show_isr-----------------------> OFF
13. selection----------------------------------> ON
14. enable_db_access---------------> OFF
15. count--------------------------> OFF
16. all_time-----------------------> OFF
17. time_trace---------------------> OFF
18. window_display-----------------> OFF
Please choose a list of switches, or type "ok."  -- [3,5,7].

Changed the switch: parse_tree-----------------------------------> ON
Changed the switch: semantics------------------------------------> ON
Changed the switch: translated_grammar_in_use--------------------> ON
yes
| ?- ssucceed.
Setting selection switch unknown_selection to -----------> succeed

yes
| ?- parse.

sentence: visual sighting of periscope followed by attack with asroc
 and torpedos.
```

Figure 1: Running PUNDIT

3 Interpreting PUNDIT Output

Syntactic processing in PUNDIT yields two syntactic descriptions of a sentence: a detailed surface structure parse tree, and an operator-argument representation called the *Intermediate Syntactic Representation*, or ISR. The ISR regularizes the information in the parse tree, reducing surface structure variants to a single canonical form and eliminating details not required for semantic analysis.

PUNDIT's semantic and pragmatic components take the ISR as input and produce a final representation of the information conveyed by the sentence which includes a decomposition of verbs into a structure of more basic predications, resolution of anaphoric references, and an analysis of temporal relations. The resulting data structure is known as the *Integrated Discourse Representation*, or IDR.

These three kinds of output will be illustrated for the following sentence:

Visual sighting of periscope followed by attack with asroc and torpedos.

This particular sentence is characteristic of the sort of input PUNDIT has been designed to handle. Note the ellipsis typical of message sublanguages[3].

3.1 The Parse Tree

The syntactic analyses produced by PUNDIT are in the formalism of String Grammar [Sager 81]. A brief glossary of String Grammar terms is provided below in figure (2) for help in understanding the parse tree in figure (3). Parse trees are displayed with siblings indented to the same depth; terminal elements (lexical items) are preceded by **==**.

3.2 The ISR

The ISR corresponding to the parse tree in figure (3) is shown in figure (4), which is taken from the output of the **parse** procedure. Two versions of the ISR are given: the first is essentially the data structure passed to semantic analysis, and the second is a pretty-printed version.

The ISR requires little knowledge of string grammar to understand. Each clause consists of syntactic operators (OPS—generally tense and aspect markers derived from the verb morphology), the verb or predicate (VERB), and its arguments. Conjunction is indicated by the insertion of the conjunction, followed by the conjuncts (set off by parallel lines). Note that each noun phrase has an associated referential index; in this example, the ISR has been printed after semantic and pragmatic analysis, and the indices have been bound to discourse entities ([sight1], [periscope1], etc.).

[3]Translation: The visual sighting of a periscope was followed by an attack (on the submarine) with anti-submarine rockets and torpedos.

`lxr`	$\Longleftrightarrow$	a left-adjunct + x + right-adjunct construction, where x can be: `n` $\Leftrightarrow$ a common noun `a` $\Leftrightarrow$ an adjective `v` $\Leftrightarrow$ a verb `ven` $\Leftrightarrow$ a past participle `tv` $\Leftrightarrow$ a tensed verb `ving` $\Leftrightarrow$ a present participle `q` $\Leftrightarrow$ a quantity word `pro` $\Leftrightarrow$ a pronoun
`nstgo`	$\Longleftrightarrow$	noun string object
`nstg`	$\Longleftrightarrow$	noun string
`sa`	$\Longleftrightarrow$	sentence adjunct
`pn`	$\Longleftrightarrow$	preposition + noun (prepositional phrase)
`tpos`	$\Longleftrightarrow$	the/determiner (prenominal) position
`qpos`	$\Longleftrightarrow$	quantity (prenominal) position
`apos`	$\Longleftrightarrow$	adjective (prenominal) position
`npos`	$\Longleftrightarrow$	noun (prenominal) position
`venpass`	$\Longleftrightarrow$	past participle + passive
`passobj`	$\Longleftrightarrow$	passive object
`nullobj`	$\Longleftrightarrow$	null object (for intransitive verb)
`thats`	$\Longleftrightarrow$	that + sentence object
`objbe`	$\Longleftrightarrow$	object of be
`vingo`	$\Longleftrightarrow$	present participle + object
`commaopt`	$\Longleftrightarrow$	comma option
`conj_wd`	$\Longleftrightarrow$	conjunction word
`spword`	$\Longleftrightarrow$	special (conjunction) word
`dstg`	$\Longleftrightarrow$	adverb string, where `d` stands for adverb.

Figure 2: A glossary of string-grammar terms

```
sentence
  center
    fragment
      zerocopula
        subject
          nstg
            infr
              ln
                apos
                  adjadj
                    larl
                      avar
                        adj == visual
              nvar
                ving == sighting
              rn
                pnpn
                  pn
                    p == of
                    nstg
                      lnr
                        ln
                        nvar
                          n == periscope
        lvr
          vvar
            null_aux
        object
          be_aux
            venpass
              lvenr
                ven == followed
              sa
                pn
                  p == by
                  nstg
                    lnr
                      ln
                      nvar
                        n == attack
                      rn
                        pnpn
                          pn
                            p == with
                            nstg
                              lnr
                                ln
                                nvar
                                  n == asroc
                                conj_wd
                                  spword == and
                                lnr
                                  ln
                                    tpos == tagged local
                                    qpos == tagged local
                                    apos == tagged local
                                    npos == tagged local
                                  nvar
                                    n == torpedos
              passobj
                nullobj
  . == .
```

Figure 3: Parse tree for *Visual sighting of periscope followed by attack with asroc and torpedos.*

```
INTERMEDIATE SYNTACTIC REPRESENTATION (ISR):

[untensed,follow,subj(passive),obj([tpos(□),[gerund,nvar([sight,singular,
[sight1]]),pp([of,[tpos(□),[nvar([periscope,singular,[periscope1]])]]])],
adj([visual])]),pp([by,[tpos(□),[nvar([attack,singular,[attack1]]),pp([with,
[and,[tpos(□),[nvar([anti^submarine^rocket,singular,[rocket1]])]],[tpos(□),
[nvar([torpedo,plural,[torpedos1]])]]]]])]]])]

OPS:   untensed
VERB:  follow
SUBJ:  passive
OBJ:   gerund: sight (sing) : [sight1]
          L_MOD:  adj: visual
          R_MOD:  pp: of
                      periscope (sing) : [periscope1]
PP:    by
       attack (sing) : [attack1]
          R_MOD:  pp: with
                              and
                      ------------------------------
                      anti^submarine^rocket (sing) : [rocket1]
                      ------------------------------
                      torpedo (pl) : [torpedos1]
```

Figure 4: ISR for *Visual sighting of periscope followed by attack with asroc and torpedos.*

3.3 The IDR

The IDR for the example sentence is shown in figure (5); its major segments are labelled **Ids**, **Properties**, **Events and Processes**, **States**, and **Important Time Relations**.

The **Ids** segment lists all the **id**, **is_group**, and **generic** predications derived during the analysis of the example sentence. **Generic** relations are established primarily to support subsequent reference through generic *they* or *one*-anaphora[4]. **Id** relations indicate the semantic type of each non-group discourse entity, while the **is_group** relations specify the semantic type, members, and cardinality of each group-level discourse entity. Thus for example the **id** relation for the entity **[sight1]**[5], derived from the nominalization *visual sighting of periscope*, indicates that the entity is an **event**, while the **is_group** relation for the entity **[projectiles1]** indicates that the entity is a group of projectiles, consisting of an unknown number of rockets and torpedos.

Relations in the **Properties** segment of the IDR are heterogeneous: these are miscellaneous relations derived in the course of processing noun phrases. Prenominal adjectives typically give rise to such relations; processing of noun-noun compounds may generate **unspecified_relationship** predications if no relationship between the nouns can be derived from domain knowledge. In the current example, the **reportingPlatform** relations are generated by a procedure which creates a default entity if the identity of the message originator is not known—if we had used the **pundit** procedure instead of **parse**, this information would have been supplied by the message header.

The **Events and Processes** and **States** segments of the IDR contain predications over discourse entities which denote situations[6]. Typically it is the processing of a clause or a nominalization which gives rise to a situation entity, and if the situation is an event, then an entity will be generated for the resulting state as well. The main predicate is the type of situation (event, state, or process), and each predication has three arguments:

1. The discourse entity
2. The associated semantic representation
3. A moment or period of time for which the situation holds

For example, the first predication in the **Events and Processes** segment in figure (5) was derived from processing the ISR for the nominalization *visual sighting of periscope*. This particular predication asserts that the referent introduced by the gerund *sighting* denotes an event; the semantic representation was constructed based on the semantics rules for the verb *sight*. All situations that are labelled events in PUNDIT can be more

[4] See [Dahl 84] for a description of the relationship between generics and one-anaphora.

[5] Labels for discourse entities are derived from the lexical head of the expression and are typically enclosed in brackets. These labels are arbitrary; **[entity2]** would do equally well.

[6] See [Passonneau 87] for a more detailed discussion of the semantics of situations.

accurately described as transitions from one state into another, where the full temporal structure of the event consists of an initial process interval, the moment of transition, and the new situation that is entered into[7]. In the second argument of the predication, the **becomeP** operator takes as its argument the semantic representation that gives rise to the new situation that is entered into, **[sight2]**. The third argument of the predication, **moment([sight1])**, should be interpreted functionally as returning the moment at which the transition into the state in question occurred. Information about this new state, **[sight2]**, is provided by a predication in the **States** field.

The final segment of the IDR lists the temporal relations which were analyzed as holding amongst the situations. Note in particular that since the verb *follow* is defined as a temporal operator, PUNDIT has correctly established the temporal relationship between the sighting and the attack.

[7]There is no referent introduced for the initial process interval of transition events.

```
Ids:
generic(torpedo)
is_group([torpedos1],members(torpedo,[torpedos1]),numb(_21227))
generic(anti^submarine^rocket)
id(anti^submarine^rocket,[rocket1])
is_group([projectiles1],members(projectile,[[rocket1],[torpedos1]]),numb(_21279))
id(us_platform,[us_platform1])
id(process,[attack1])
generic(periscope)
id(periscope,[periscope1])
id(us_platform,[us_platform3])
id(state,[sight2])
id(event,[sight1])

Properties:
reportingPlatform([us_platform1])
reportingPlatform([us_platform3])

Events and Processes:
event(
  [sight1]
  becomeP(sightP(experiencer([us_platform3]),theme([periscope1]),instrument(visual)))
    sighted_atP(theme([periscope1]),location(_28507))
  moment([sight1]))

process(
  [attack1]
  doP(attackP(actor([us_platform1]),theme(_19607),instrument([projectiles1])))
  period([attack1]))

States:
state(
  [sight2]
  sightP(experiencer([us_platform3]),theme([periscope1]),instrument(visual))
    sighted_atP(theme([periscope1]),location(_28507))
  period([sight2]))

Important Time Relations:
the sight state ([sight2]) started with the sight event ([sight1])
the sight event ([sight1]) preceded the arbitrary event time (moment([attack1]))
of the attack process ([attack1])
```

Figure 5: IDR for *Visual sighting of periscope followed by attack with asroc and torpedos.*

4 Commonly Used Procedures

4.1 edit_rule

The procedure **edit_rule/1** allows you to edit a set of grammar rules for a specified non-terminal, using the Prolog Structure Editor. For more details, please consult [Riley 86].

4.2 edit_word

The procedure **edit_word/1** allows you to edit the lexical entry for a specified word, using the Prolog Structure Editor. For more details, please consult [Riley 86].

4.3 parse

The procedures **parse** and **pundit** (see below) provide two slightly different front-ends to the PUNDIT system. **parse** is the access method of preference for those whose primary interest is parsing or minimizing keystrokes (no prompts are issued to collect message header information). The **parse** procedure is a core component of PUNDIT, and is domain-independent.

The behavior and output of **parse** are largely controlled by switch settings (see Section 4). Briefly, the **parse** procedure collects the input to be analyzed by PUNDIT, and then calls syntactic analysis. Depending on your switch settings, it may then call semantic analysis, the database extractor, and the summary module (if defined for the current domain). Depending again on switch settings, you may be shown both intermediate and final results: trace messages, the parse trees, the ISRs, the IDR, database relations extracted, and a summarization of the input text[8]. In the course of processing your input, PUNDIT may engage you in dialogue if certain switches are turned on: for example, the Selection module may ask you about co-occurrence patterns; if the switch **enter_new_word** is on, you will be prompted to enter lexical information for new words.

The initial prompt to collect the input depends on switch settings as well. If the switch **text_mode** is on, you will be prompted to enter a paragraph of text: that is, one or more sentences followed by two carriage returns[9]. In this case, the input will be processed one sentence at a time, and the first parse for each sentence will be processed.

If the switch **text_mode** is off, you will be prompted to enter a single sentence; after processing the first parse, you will be invited to continue with the next parse, until you wish to stop or all parses have been exhausted.

[8] The summary application is not implemented in the MUCK domain.

[9] Since each sentence may optionally be followed by one carriage return, the extra carriage return at the end is needed to signal the end of input. Moreover, although PUNDIT will process run-on sentences (without punctuation), the final sentence must have a terminator: a period, exclamation point, or question mark.

In addition to these capabilities, designed for the processing of sentences, you may also analyze lower-level constituents. To process an isolated noun phrase, call `parse_np/0` (this procedure supports both syntactic and semantic analysis). NPs and other constituents may also be parsed by invoking `parse/1`, giving as argument the grammatical category (this will require a knowledge of PUNDIT's grammatical categories). As a simple illustration, you may parse the noun phrase *visual sighting of periscope* by calling `parse(lnr)`. Note, however, that `parse(lnr)` does not support semantic analysis.

4.4 pundit

The `pundit` procedure provides a domain-specific front-end to the PUNDIT system, one geared specifically towards full message processing. Since `pundit` is similar in many respects to `parse` (see above), only differences will be described here.

First, `pundit` is not sensitive to the `semantics` and `text_mode` switches: it is assumed that all messages require semantic analysis, and that all input will be one or more sentences of text. As a result, it is not possible to request multiple parses of the input. However, if a sentence fails semantic analysis, `pundit` will backtrack for the next parse, and this process will continue until a semantically acceptable parse is found.

Secondly, `pundit` provides a domain-specific message entry screen which collects the message header and the message body. The screen for the MUCK domain is shown in Figure (6) below (you may enter a question mark at any prompt to receive a description of valid responses). The responses to the first four prompts are used to establish the discourse context for the interpretation of the message body.

The `pundit` procedure also provides capabilities for processing one or more existing messages from the message corpus (stored in `<domain>_working.pl`). When you first invoke `pundit`, the message corpus is compiled into your image, creating entries in the recorded database[10]. At the prompt for `Message number`, you may enter the number of an existing message, and `pundit` will fetch the message from the recorded database and process it. If you wish to process a list of existing messages, call `pundit(batch,YourList)`, where YourList is a Prolog list of message numbers. You may also process the entire message corpus by calling `pundit(batch,test_pundit)`[11].

[10] If there is a version of the message corpus in your directory, pundit will load that; otherwise, it will load the file from the main domain directory. This feature allows you to maintain a personal corpus of texts.

[11] This is the method which we use to test software changes: the output can be saved in a file and compared against the results of testing a previous image.

```
%~nlp/pundit/muck/Muck.qimage +

Loading /usr/local/bin/em215 with /mn2/q2.2/ml...
Unix Prolog+Emacs V2.15 (01-Jan-88)
Copyright(c) 1986, 1987 Unipress Software, Inc.

Quintus Prolog Release 2.2 (Sun-3, Unix 3.2)
Copyright (C) 1987, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

 [consulting /mn2/cball/prolog.ini...]
Setting selection switch unknown_selection to ------------> succeed
 [prolog.ini consulted 0.133 sec 720 bytes]

| ?- pundit.

[compiling /nlp/nlp/pundit/muck/muck_working.pl...]
[muck_working.pl compiled 2.700 sec 12,612 bytes]

************************* RAINFORM MESSAGE ENTRY *********************

Message number            [1] :11
Enemy platform       [barsuk] :submarine
Reporting platform [virginia] :texas
Report time           [0800t] :0800t

Sighting message: sighted periscope an asroc was fired proceeded to
station visual contact lost, constellation helo hovering in vicinity.
sub appeared to be ooa.

     Processing discourse segment...

..

     Segment processing Time: 39.967 sec.


****************************** Complete IDR ****************************

(etc.)
```

Figure 6: Using the pundit procedure

4.5 punt

This procedure provides on-line documentation for several PUNDIT utilities: the *Prolog Structure Editor*, the *Lexical Entry Procedure*, tools for creating a concordance, and the *Dictionary Merge* utility. To invoke the punt utility, type punt at the Prolog prompt.

4.6 rdb_remove

This development utility removes entries of specified type(s) from the Prolog recorded database. It is useful when testing changes to one of the files whose compilation creates such entries. For example, the pundit procedure, as one of its steps, compiles the message corpus into your current image. If you should wish to edit and reload the message file (<domain>_working.pl), you must first remove the old messages: rdb_remove facilitates this task. A sample session is given below.

```
| ?- rdb_remove.

Recorded Database Rules:
1. The Lexicon (dict)
2. The Bnf (bnf)
3. Define and Simplification Rules (define)      [obsolete]
4. Semantic Selection Rules (semantics)          [obsolete]
5. Clause Mapping Rules (mapping)                [obsolete]
6. Noun Phrase Mapping Rules (mapping_np)        [obsolete]
7. All Semantics Rules (all_semantics)           [obsolete]
8. The Selectional Patterns (selection)
9. The Stable Messages (messages)
10. quit
Please choose a list of items --  [9].

Erasing corpus muck...

Time to erase the testing messages: 0.15 sec.
```

Figure 7: Using the rdb_remove utility

Note that options 3-7 are obsolete (semantics rules are not stored in the recorded database).

4.7 readIn

The procedure **readIn/1** loads a PUNDIT lexicon into the current image. Its argument is the name of a lexicon file. For example, to load the lexicon file **my_lex.pl** from the current working directory, execute the goal **readIn(my_lex)**. Lexical entries are stored in the recorded database; to avoid duplicate entries, it may be necessary to run **rdb_remove** to remove previous entries before using **readIn** to load a new lexicon.

4.8 squery

The predicate **squery/0** is used to control the behavior of the Selection component when it encounters an unknown selectional pattern. Execute the goal **squery** to be queried when an unknown pattern is encountered. For more details, see Section 12 of [Lang 87].

4.9 ssucceed

The predicate **ssucceed/0** is analogous to **squery/0**, except that it is used to allow unknown selectional patterns to succeed. There is also a predicate **sfail/0** which can be used to force unknown selectional patterns to fail. For more details, see [Lang 87].

4.10 switches

The **switches** utility allows you to control the operation of PUNDIT. Each switch and its dependencies are described in more detail below.

```
| ?- switches.

1. enter_new_word------------------> OFF
2. np_trace------------------------> OFF
3. parse_tree----------------------> OFF
4. conjunction---------------------------------> ON
5. semantics-----------------------> OFF
6. translated_grammar_present------------------> ON
7. translated_grammar_in_use-------> OFF
8. grinder-------------------------> OFF
9. text_mode-----------------------> OFF
10. decomposition_trace------------> OFF
11. summary------------------------> OFF
12. show_isr-----------------------> OFF
13. selection----------------------------------> ON
14. enable_db_access---------------> OFF
15. count--------------------------> OFF
16. all_time-----------------------> OFF
17. time_trace---------------------> OFF
18. window_display-----------------> OFF
Please choose a list of switches, or type "ok."  -- [5,7,9].


Changed the switch: semantics--------------------------------> ON

Changed the switch: translated_grammar_in_use----------------> ON

Changed the switch: text_mode--------------------------------> ON
```

Figure 8: Using the **switches** utility

Several related procedures are useful in this connection. The procedure **status** displays current switch settings; **flip/1** reverses the setting of one switch (for example, **flip(semantics)**); **turn_on/1** and **turn_off/1** turn a specified switch on and off.

4.10.1 enter_new_word

This switch controls the behavior of PUNDIT when lexical lookup encounters a word which is not in the lexicon and which cannot be analyzed by the Shapes module. If the input to PUNDIT contains an unrecognizable word and this switch is **off**, lexical lookup will issue the following error message:

No definition found for -- *<UNKNOWN-WORD>*

sentence failed ...

If the switch is **on**,you will be given the following options:

1. Respell word
2. Add dictionary entry
3. Word is a proper noun
4. Quit

Choose the first option if you have simply misspelled the word. If the word is a proper name, you may choose the third option (but no dictionary entry will be created). If you choose to add a new dictionary entry, the *Lexical Entry Procedure* is invoked, and you will be prompted to enter morphological and grammatical information, which may be optionally saved in a file in your directory (consult [Riley 88] and [Linebarger 88] for more detail). Note that the information collected will allow PUNDIT to proceed with the syntactic analysis of the input, but may not be sufficient to enable semantic analysis: for this, it may be necessary to add new semantics rules and/or update the knowledge base.

4.10.2 np_trace

This switch controls the display of Reference Resolution trace messages concerning the creation of discourse entities. Turning this switch on will only have an observable effect if the **semantics** switch is turned on as well.

4.10.3 parse_tree

This switch controls printing of the parse tree and the ISR. The parse tree and ISR are always computed whether this switch is on or not.

4.10.4 conjunction

This switch is one of several switches that cannot be switched. The switch will be **on** if the conjunction meta-rule has been applied to the grammar, and will be **off** otherwise. If this switch is off, and you want the grammar to include conjunction, run the procedure **gen_conj/0**. After the meta-rule has been applied, the switch will automatically be turned on. Since the meta-rule cannot be undone, the switch cannot subsequently be turned off.

4.10.5 semantics

Turn this switch **on** to enable semantic and pragmatic analysis of input; turn it **off** if you wish only to parse. Only the **parse** procedure is sensitive to this switch: the **pundit** procedure assumes that you want a full analysis of the input.

4.10.6 translated_grammar_present

The switch indicates whether or not the grammar has been translated into Prolog. The switch is **on** in the software which accompanies this document, and cannot be turned off.

If at your site an image has been developed in which this switch is **off**, then the grammar must be run interpreted. Running interpreted is slow, but it facilitates debugging and rapid grammar changes. Turning the switch on will translate the grammar, which may take a few minutes; after translation, you will be given the option to compile the resulting Prolog code. You will normally want to do this, because the compiled translated grammar provides the fastest parsing. The only reason not to do this is if you want to use the Prolog debugger on the translated code, which is not advised. If at any time you want to compile the translated grammar, compile the file **translated_grammar.pl**.

4.10.7 translated_grammar_in_use

This switch allows you to parse with the grammar translated (**on**) or interpreted (**off**). Although the switch is off in the software which accompanies this document, you will normally want it to be on (for the fastest parsing). The only reason to turn this switch off is to make use of certain grammar debugging tools that are only available when interpreting the grammar, such as **grinding** and **counting**.

4.10.8 grinder

This switch allows you to trace the application of grammar rules and restrictions, a development feature which is only available when parsing with the grammar interpreted (if

you turn this switch **on**, the **translated_grammar_in_use** switch will automatically be turned off).

The facility is called *grinder* because it typically produces considerable output. To reduce the amount of output, you may choose to trace only the application of specific grammar rules or restrictions.

```
| ?- turn_on(grinder).

Enter one of:  [<what you want to grind on>],
               off,  or
               all
 ** WARNING ** If you grind at all, you will automatically run interpreted.
Enter choice:
```

Figure 9: Setting the **grinder** switch

4.10.9 text_mode

This switch is used by the procedure **parse**. If it is **on**, you will be prompted to enter a paragraph of text (one or more sentences followed by two carriage returns). Only the first parse for each sentence in the paragraph will be processed. If the switch is **off**, you will be prompted to enter a single sentence, and you may step through all parses for that sentence.

4.10.10 decomposition_trace

This switch allows you to monitor the course of semantic analysis: if it is **on**, a variety of trace messages will be displayed, including the ISR for each clause about to be processed and the semantic representation of the input as it is built up. While the switch was designed to facilitate development of semantics rules and the knowledge base, the trace messages are also useful when diagnosing the source of an incorrect or unsuccessful semantic analysis. Note that **decomposition_trace** has no effect unless the **semantics** switch is also on.

4.10.11 summary

This switch controls whether or not a domain-specific module is called to create a summary of the input text. Since summaries depend on the output of semantic analysis, the **semantics** switch must be turned on. Note: the summary application has not been implemented in the MUCK domain.

4.10.12 show_isr

This switch controls the display of the ISR; its effect depends on whether you are using **parse** or **pundit**. If the switch is **on** and you are using the **parse** procedure, the incremental ISR will be displayed for each node in the parse tree. This is useful for debugging changes to the ISR, but not recommended otherwise. Note that the **parse_tree** switch must also be on in this case (when using **parse**, you cannot see the ISR without also displaying the parse tree).

If you are using the **pundit** procedure and this switch is **on**, the ISR for each sentence will be displayed after syntactic analysis and before semantic analysis. In this case, the **parse_tree** switch need not be on.

4.10.13 selection

This switch controls whether or not the Selection module is invoked in the course of parsing. If it is **on**, Selection will be called; if it is **off**, Selection will not be called. For more details, see [Lang 87].

4.10.14 enable_db_access

This switch controls whether or not queries and assertions access the database defined for the current domain. It is used by the procedures **parse** and **pundit**. If the switch is **on**, domain-specific database definitions will be used to extract database relations from the results of semantic analysis, and these relations will be displayed on your screen.

Dependencies: **semantics** must be turned **on**, and database relations must be defined for the current domain (<domain>_db_structure.pl and <domain>_db_mapping.pl).

4.10.15 count

This switch should be left off.

4.10.16 all_time

This switch controls the display of the time relations segment of the IDR. If it is **off**, the segment is labelled **Important Time Relations** and contains what are judged to be the most prominent temporal relations discovered during temporal analysis of the input. If it is turned **on**, the segment is labelled **Complete Time Relations**, and all the relations that could be discovered are displayed. Turning this switch on will only have an observable effect if the **semantics** switch is turned on as well.

4.10.17 time_trace

This switch allows you to monitor the course of temporal analysis. If it is **on**, informative trace messages will be displayed about situation representations as they are constructed by the Time component. Turning this switch on will only have an observable effect if the **semantics** switch is turned on as well.

4.10.18 window_display

This switch should be left off.

A Installing the System

The PUNDIT system runs under release 4.3 of Berkeley UNIX and Quintus Prolog (currently release 2.2). Before installing PUNDIT, a /nlp partition should first be created; this partition should contain the directory /nlp/nlp/pundit, where the core PUNDIT components will be installed. Software for the MUCK domain will be installed in the /nlp/nlp/pundit/muck subdirectory.

If these partitions and directories cannot be created, several absolute path names in PUNDIT code will require modification: the files and lines of code are listed below. Note that if it is necessary to create alternative directories to those recommended, please ensure that core PUNDIT files and domain-specific files are stored in separate directories.

FILENAME	code
punt.pl	`:- asserta(home_dir("/nlp/nlp/pundit/")).`
qprolog15.pl	`timeCom :- unix(shell('/mn2/AI/nlp/bin/timeCom')).`
sem_edit.pl	`:- compile('~nlp/pundit/semed/correctForms.pl').`
switches.pl	`compile('~nlp/pundit/count_on.pl').`
switches.pl	`compile('~nlp/pundit/count_off.pl').`
compilePundit	`pundit_directory('/nlp/nlp/pundit').`
compileMuck	`muck_directory('/nlp/nlp/pundit/muck').`

We strongly recommend that the files in the PUNDIT home directory (and its subdirectories) be owned by a special user, and that the file protections be set in such a way that only this special user can alter these files.

B Building PUNDIT Images

B.1 Building a Core PUNDIT Image

To create a core PUNDIT image, execute the following sequence of steps:

1. go to a directory to which you have write permission
2. type to the UNIX prompt the command
   qprolog2.2 < /nlp/nlp/pundit/makePundit[12]

Executing these steps will deposit in the current working directory a Prolog saved state called **Pundit.testimage**, which is the core PUNDIT image.

B.2 Creating a Functional Core PUNDIT Image

The core PUNDIT image itself is not functional (i.e., it cannot be used to parse sentences), and is only used to build the domain-specific images. If, however, a user wishes to make a functional image from a core PUNDIT image, the following steps should be executed:

- Create a file containing the following Prolog code:

```
% ------------------------------------------------------------

% Turn on conjunction and translate the grammar
:- gen_conj.
:- translate_grammar('/nlp/nlp/pundit/translated_grammar.pl').
:- compile('/nlp/nlp/pundit/translated_grammar.pl').
:- compile('/nlp/nlp/pundit/muck/compute_types.pl').

% These declarations are required for the Selection module
pundit_domain(core).

isa(nothing,nothing).

semantic_type(nothing,nothing).

% ------------------------------------------------------------
```

[12] Instead of qprolog2.2, you should use whatever command is necessary at your site to start up the current version of Quintus Prolog.

- Start up the core PUNDIT image and compile the file containing the code above.
- Save the resulting image (e.g., by executing the goal `save_program('Pundit.newimage')`.

Note that this image can be used only for parsing, since most of the procedures required for semantic analysis (e.g. the knowledge base and semantics rules) are domain-specific.

B.3 Creating a Complete Domain-Specific Image

To create a complete domain-specific image (in this case, an image for the MUCK domain), follow these steps:

- again, go to a directory to which you have write permission
- type to the UNIX prompt the command
  `/nlp/nlp/pundit/Pundit.testimage < /nlp/nlp/pundit/muck/makeMuck`.[13]

Executing these steps will deposit in the current working directory a Prolog saved state called `Muck.testimage`, which is the complete domain image. Once the above procedure has been completed, either of these two Prolog saved states can be started up simply by typing `Pundit.testimage` or `Muck.testimage` to the UNIX prompt (or by typing the absolute filename, if the user is not in the directory in which these files are found). The images can, of course, be renamed if desired.

[13] This assumes that `Pundit.testimage` is currently in the directory `/nlp/nlp/pundit`.

C Customizing Your PUNDIT User Environment

Because PUNDIT is written in Quintus Prolog, we can use one of its features to make it easy to customize PUNDIT for individual use. When Prolog first starts up, it checks in the user's home directory for a file named `prolog.ini`. If such a file exists, Prolog will compile it into its current image. Using this feature, we can instruct Prolog to automatically set PUNDIT switches to those settings that we find most convenient. In Figure 10 is an example of one such `prolog.ini`. The example code first checks to see if Prolog is running a PUNDIT image; if it is, switches are set to the desired settings (in this case, to those most convenient for grammar development). Observe in particular that the switch `translated_grammar_in_use` is turned on only if `translated_grammar_present` is already on. At the end, a procedure is called which displays the current switch settings.

```
turn_on_initial_switches:-
    recorded(toggle,switches_are_defined,_),
    !,
    (toggle(translated_grammar_present)->
        turn_on(translated_grammar_in_use);
        true),
    turn_on(parse_tree),
    turn_off(selection),
    ssucceed,
    turn_off(show_isr),
    turn_off(semantics),
    turn_off(text_mode),
    turn_off(summary),
    show_herald.

turn_on_initial_switches.

:- turn_on_initial_switches.
```

Figure 10: Sample `prolog.ini` file

D PUNDIT Files and Dependencies

D.1 Files

Listed below are the core and domain-specific files which comprise the PUNDIT software accompanying this document. By convention, domain-specific files are prefixed with the name of the domain.

- Core Files
  - Lexical
    - **dictisr.pl** - the core lexicon
    - **entries.pl** - the Lexical Entry Procedure
    - **lookup.pl** - lexical lookup
    - **reader.pl** - procedures to read input
    - **readin.pl** - load or update the lexicon
    - **shapes.pl** - shape descriptors
    - **tables.pl** - lexical entry options
  - Syntax
    - Grammar
      - **bnf.pl** - bnf definitions
      - **compile_types.pl** - [created automatically]
      - **compute_types.pl** – compute atomic grammar nodes
      - **conj_restr.pl** - grammar restrictions for conjunction
      - **count_off.pl** - counting procedure
      - **count_on.pl** - counting procedure
      - **counting.pl** - procedures for grinding and counting
      - **interpreter.pl** - grammar interpreter
      - **lspops.pl** - elementary restriction operators
      - **meta.pl** - meta grammar for conjunction
      - **path.pl** - navigate the parse tree
      - **prune.pl** - dynamic pruning of grammar options
      - **restrictions.pl** - restrictions
      - **routines.pl** - basic syntactic routines for grammar
      - **translated_grammar.pl** - [created automatically]
      - **translator.pl** - grammar translator
      - **types.pl** - type definitions for grammar
      - **update.pl** - grammar update procedures

      - **xor.pl** - *exclusive or* mechanism for grammar options
    * Intermediate Syntactic Representation
      - **compute_trans.pl** - compute ISR
      - **isr_lexical.pl** - ISR information for terminal symbols
      - **isr_ops.pl** - ISR operator definitions
      - **semproc.pl** - simplify ISR translation
      - **show_isr.pl** - display procedures for the ISR
    * Selection
      - **selection_dcg.pl** - Selection DCG for analyzing ISR
      - **selection_query.pl** - Selection user interface
      - **selection_restr.pl** - restrictions which call Selection DCG
      - **selection_tools.pl** - Selection tools
      - **selection_top_level.pl** - record and erase parsed sentences
      - **selection_utilities.pl** - Selection utilities
  - Semantics
    * **adjunct_analysis.pl** - analyze sentence adjuncts
    * **filter.pl** - prepare ISR for semantic analysis
    * **np_int.pl** - noun phrase semantics
    * **quantifiers.pl** - quantifier binding procedures
    * **semantics.pl** - the Semantic Interpreter
    * **world.pl** - general knowledge base procedures
  - Pragmatics
    * **discourse_rules.pl** - manage discourse and focus information
    * **np_ext.pl** - Reference Resolution
    * **time.pl** - Time Analysis
  - Database Application
    * **entry_generator.pl** - create database relations
  - Utilities
    * **access.pl** - ISR accessor functions
    * **edit.pl** - Prolog Structure Editor
    * qprolog15.pl - code specific to Quintus Prolog
    * **rdb_remove.pl** - remove entries from recorded database
    * **show.pl** - display ISR, IDR, db relations, etc.
    * **switches.pl** - manage PUNDIT switches
    * **testing.pl** - software testing utility (not for MUCK
    * **time_display.pl** - temporal relations display procedures

        * **trace_messages.pl** - semantics trace messages
        * **utilities.pl** - general-purpose procedures
        * **vax_menus.pl** - menu facility
        * **vax_show.pl** - top-level non-window display procedures
        * **ws_support.pl** - windowing system procedures
    - Other
        * **compilePundit** - build a PUNDIT image
        * **demo_top_level.pl** -
        * **op_defs.pl** - operator declarations
        * **punt.pl** - on-line PUNDIT help
        * **top_level.pl** - PUNDIT front-end

- Domain-Specific Files for the MUCK Domain
    - Lexical
        * **muck_dictisr.pl** - incremental lexicon
        * **muck_shapes.pl** - shape descriptors
    - Syntax
        * Grammar
            · **compile_types.pl** - [created automatically]
            · **muck_bnf.pl** - updates to the core bnf file
            · **muck_restrictions.pl** - restrictions
            · **translated_grammar.pl** - [created automatically]
        * Selection
            · **muck_selection_db.pl** - selectional patterns
            · **SELECTIONAL_PATTERNS.pl** - [created automatically by Selection]
            · **USER_CORPUS.pl** - [created automatically by Selection]
    - Semantics
        * **muck_rules.pl** - semantics rules
        * **muck_world.pl** - the knowledge base
    - Pragmatics
        * **muck_time.pl** - temporal operators and rules
    - Database Application
        * **muck_entry_generator.pl** - customized version of core file
        * **muck_db_structure.pl** - database definition
        * **muck_db_mapping.pl** - database mapping
    - Summary Application

    * **muck_summary.pl** - create summaries (empty file)
  - Other
    * **compileMuck** - build MUCK image
    * **muck_top_level.pl** - message entry front-end
    * **muck_working.pl** - message corpus

D.2 Dependencies

While most PUNDIT files can be loaded in any order, certain files and classes of files must be loaded in a specific order for PUNDIT to run correctly. These ordering dependencies arise for three main reasons:

1. Compilation of domain-specific files is designed to follow compilation of domain-independent files. For example, certain core procedures may be abolished and redefined in a domain-specific file; if changes are made to the core file and it is recompiled in a domain image, the domain-specific file must be recompiled as well.
2. Some of PUNDIT's data are stored in the Prolog internal database, and multiple compilations of certain files will result in duplicate database entries. The relevant files are: the core and domain-specific versions of the grammar and the lexicon (**bnf.pl** and **dictionar.pl**), and the domain selectional patterns and message corpus.
3. Certain operations in PUNDIT are performed at compile time. These include meta-rules for the grammar, translating the grammar, and computing the types of non-terminals in the grammar. These operations must be done in order.

If, in the course of development, you wish to compile a new version of the grammar, lexicon, selectional database or message corpus, you must first remove the internal database entries generated by the compilation of the previous version. This can be done most simply by calling the procedure **rdb_remove** (see Section 4), which removes *all* database entries of a specified type.

Compiling changes to selectional patterns: selectional patterns reside in two files: <domain>_selection_db.pl and SELECTIONAL_PATTERNS.pl. The latter is created automatically in any directory in which you have run a PUNDIT image with the **selection** switch on, while the former resides in the main domain directory, is maintained by hand, and is compiled into the standard domain image. If you wish to retain the selectional patterns which were originally compiled into the image and to add your personal selectional patterns, compile <domain>_selection_db.pl and SELECTIONAL_PATTERNS.pl, in that order. Otherwise, compile only the relevant file.

Compiling changes to the message corpus: the message corpus is not compiled into either the core PUNDIT image or the domain image; instead, it is automatically compiled

into your image when you first invoke the `pundit` command. Therefore, if you have modified this file, you need not recompile it yourself. The system supports personal versions of the corpus: if the file `<domain>_working.pl` exists in the directory in which you are running an image, that is the file which will be compiled. If it does not exist, the file in the main domain directory will be compiled.

Loading changes to the lexicon: multiple lexicon files exist. The core PUNDIT lexicon (`dictisr.pl`) resides in the core PUNDIT directory and is incorporated into the core PUNDIT image; the domain-specific lexicon (`<domain>_dictisr.pl`) resides in the domain directory and is incorporated into the domain image. Since domain images are built from core images, a domain image contains lexical entries from *both* the core lexicon and the domain lexicon, loaded in in that order. In addition, you may have one or more personal lexicon files created by using the Lexical Entry Procedure. By running `rdb_remove` to remove lexical entries, you will have removed *all* lexical entries, regardless of the file in which they originated. You will now need to use the `readIn` procedure, and load the relevant lexicon files in sequence.

Implementing changes to the grammar:

1. Read in new grammar file
2. Meta-Rules—run `gen_conj/0`.
3. Translate the grammar to Prolog—run `translate_grammar/1`, whose argument is a file name (generally `translated_grammar.pl`).
4. Compile the translated grammar—compile the file named above.
5. Compute the types of the grammar nonterminals—compile the file `compute_types.pl`.

These steps must be performed in the order listed, except that step 5 may be performed any time after step 2. Step 2 may be skipped if you do not wish to parse sentences containing conjunction. Skip *both* steps 3 and 4 if you wish to parse with the grammar interpreted (at a significant performance loss). Generally speaking, you will always need to recompile `compute_types.pl`.

Compiling changes to files which do not update the recorded database : certain files exist in core and domain-specific versions (e.g. `shapes.pl` and `<muck>_shapes.pl`). The core versions reside in the core PUNDIT directory and are incorporated into the core PUNDIT image; the domain-specific versions reside in the domain directory and are incorporated into the domain image. Since the domain image is built from the core image, domain-specific files are compiled on top of core files. If you are working in a domain image and have changed a file which exists in both core and domain-specific versions, you will need to recompile both, in that order. Otherwise, simply recompile the relevant file.

E PUNDIT Bibliography

E.1 Background Reading

Dahl, Deborah A. *The Structure and Function of One-Anaphora in English.* PhD thesis, University of Minnesota, 1984; Indiana University Linguistics Club, 1985.

Hirschman, L. Discovering Sublanguage Structures. In Kittredge, R. and Grishman, R. (editors), *Sublanguage: Description and Processing.* Lawrence Erlbaum Assoc., Hillsdale, NJ, 1986.

Palmer, Martha. *Driving Semantics for a Limited Domain.* PhD thesis, University of Edinburgh, 1985.

Palmer, Martha S. *Semantic Processing for Finite Domains.* To appear as a volume in *Studies in Natural Language Processing,* Cambridge University Press, editor, Aravind Joshi, 1988.

Sager, Naomi. *Natural Language Information Processing: A Computer Grammar of English and Its Applications.* Addison-Wesley, 1981.

E.2 Papers and Presentations

Dahl, Deborah A. Focusing and Reference Resolution in PUNDIT. In *Proceedings of the 5th International Conference on Artificial Intelligence.* Philadelphia, PA, August 1986.

Dahl, Deborah A. Determiners, Entities, and Contexts. In *Proceedings of TINLAP-3.* Las Cruces, NM, January 1987.

Dahl, Deborah, Dowding, John, Hirschman, Lynette, Lang, François, Linebarger, Marcia, Palmer, Martha, Passonneau, Rebecca, and Riley, Leslie. *Integrating Syntax, Semantics, and Discourse.* Darpa Natural Language Understanding Program. R&D Status Report, Unisys Defense Systems, May 14, 1987.

Dahl, Deborah A. Integration of Semantics and Pragmatics in the Computational Analysis of Nominalizations. Colloquium presented to the Department of Computer Science, The Pennsylvania State University, October, 1987.

Dahl, Deborah A., Palmer, Martha S., and Passonneau, Rebecca J. Nominalizations in PUNDIT. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics.* Stanford University, Stanford, CA, July 1987.

Dahl, D. Natural Language Processing for Database Generation: The PUNDIT System. Paper presented at AI West, May, 1988, Long Beach California.

Dowding, John and Hirschman, Lynette. Dynamic Translation for Rule Pruning in Restriction Grammar. In *Proceedings of the 2nd International Workshop on Natural Language Understanding and Logic Programming.* Vancouver, B.C., Canada, 1987.

Grishman, R. and Hirschman, L. PROTEUS and PUNDIT: Research in Text Understanding. *Computational Linguistics* 12(2):141-45, 1986.

Hirschman, Lynette. Conjunction in Meta-Restriction Grammar. *Journal of Logic Programming* 4:299-328, 1986.

Hirschman, Lynette. Natural Language Interfaces for Large Scale Information Processing. Technical Advisory Panel Meeting for the Transportation Systems Center, Department of Transportation. Boston, MA, May, 1987.

Hirschman, Lynette, Tutorial on Natural Language and Logic Programming. 1987 Logic Programming Symposium, San Francisco, Aug. 31-Sept. 4, 1987.

Hirschman, L. A Meta-Treatment of wh-Constructions. To be presented at META 88 Workshop on Meta Programming in Logic Programming.

Hirschman, Lynette, Dahl, Deborah, Dowding, John, Lang, François-Michel, Linebarger, Marcia, Palmer, Martha, Riley, Leslie, and Schiffman, [Passonneau] Rebecca. The PUNDIT Natural Language Processing System. Presented at the Eleventh Annual Penn Linguistics Colloquium, Philadelphia, PA, February, 1987.

Hirschman, L., Hopkins, W.C., Smith, R.C. Or-Parallel Speed-up in Natural Language Processing: A Case Study. To be presented at the 5th International Logic Programming Conference, Seattle, August, 1988.

Hirschman, L. and Puder, K. Restriction Grammar in Prolog. In *Proceedings of the First International Logic Programming Conference,* pages 85-90.

Hirschman, L. and Puder, K. Restriction Grammar: A Prolog Implementation. In Warren, D.H.D. and Van Caneghem, M. (editors), *Logic Programming and its Applications,* pages 244-261. Ablex Publishing Corp., Norwood, NJ, 1986.

Lang, François-Michel and Hirschman, Lynette. Improved Portability and Parsing through Interactive Acquisition of Semantic Information. In *Proceedings of the Second Conference on Applied Natural Language Processing.* Austin, TX, February 1988.

Linebarger, Marcia C., Dahl, Deborah A., Hirschman, Lynette, and Passonneau, Rebecca J. Sentence Fragments Regular Structures. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics.* Buffalo, NY, June 1988.

Palmer, Martha S., Dahl, Deborah A., Passonneau, Rebecca J., Hirschman, Lynette, Linebarger, Marcia, and Dowding, John. Recovering Implicit Information. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics.* Columbia University, New York, August 1986.

Palmer, Martha, and Linebarger, Marcia. Status of Verb Representations in PUNDIT. Presented at Theoretical And Computational Issues in Lexical Semantics, Brandeis University, Waltham, Mass, April 21-24, 1988.

Palmer, Martha, Hirschman, Lynette, and Dahl, Deborah. Text Processing Systems. June 1988. Tutorial presented at the 26th Annual Meeting of the Association for Computational Linguistics, Buffalo New York.

Passonneau, Rebecca J. Situations and Intervals. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 16-24. 1987.

Passonneau, Rebecca J. A Computational Model of the Semantics of Tense and Aspect. *Computational Linguistics* (forthcoming), 1988.

E.3 Technical Documentation

Ball, Catherine N., Dahl, Deborah A., Dowding, John, Hirschman, Lynette, Linebarger, Marcia, Palmer, Martha, and Passonneau, Rebecca. *PUNDIT Tutorial Notes.* Internal document, Unisys Corporation, 1987.

Lang, François-Michel. *A User's Guide to the Selection Module.* LBS Technical Memo 68, Unisys Corporation, 1987.

Linebarger, Marcia C. *A Guide to Object Options in PUNDIT.* Technical Report, Unisys Corporation, 1988.

Riley, Leslie. *A Guide to the PUNDIT Lexical Entry Procedure.* Technical Report, Unisys Corporation, 1988.

Riley, Leslie and Dowding, John. *The Prolog Structure Editor.* LBS Technical Memo 29, Unisys Corporation, 1986.

Schiffman (Passonneau), Rebecca J. *Designing Lexical Entries for a Limited Domain.* LBS Technical Memo 42, Unisys Corporation, April 1986.

References

[Dahl 84] Dahl, Deborah A. *The Structure and Function of One-Anaphora in English.* PhD thesis, University of Minnesota, 1984.

[Lang 87] Lang, François-Michel. *A User's Guide to the Selection Module.* LBS Technical Memo 68, Unisys Corporation, 1987.

[Linebarger 88] Linebarger, Marcia C. *A Guide to Object Options in PUNDIT.* Technical Report, Unisys Corporation, 1988.

[Passonneau 87] Passonneau, Rebecca J. Situations and Intervals. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 16–24. 1987.

[Riley 86] Riley, Leslie and Dowding, John. *The Prolog Structure Editor.* LBS Technical Memo 29, Unisys Corporation, 1986.

[Riley 88] Riley, Leslie. *PUNDIT Lexical Entry Procedure User's Guide.* Technical Report, Unisys Corporation, 1988.

[Sager 81] Sager, Naomi. *Natural Language Information Processing: A Computer Grammar of English and Its Applications.* Addison-Wesley, 1981.

PUNDIT
Lexical Entry Procedure
User's Guide*

Version 1.1

November 21, 1988

Unisys Logic-Based Systems
Paoli Research Center
P.O. Box 517, Paoli, PA 19301

*This work has been supported by DARPA contract N00014-85-C-0012, administered by the Office of Naval Research.

Contents

1 Introduction 1

1.1 Features . . . 1

1.2 Limitations . . . 1

2 Getting Started 1

2.1 How to access the LEP . . . 1

2.2 General conventions . . . 3

2.2.1 Meta-responses . . . 3

2.2.2 Defaults . . . 3

2.2.3 Menus . . . 3

3 Defining New Words 5

3.1 Before You Begin . . . 5

3.2 Initial Prompts . . . 5

3.3 Word Classes . . . 7

3.3.1 Nouns . . . 7

3.3.2 Proper Nouns . . . 8

3.3.3 Verbs . . . 9

3.3.4 Adjectives . . . 11

3.3.5 Adverbs . . . 11

3.3.6 Determiners . . . 11

3.3.7 Quantifiers . . . 12

3.3.8 Prepositions . . . 12

3.4 Completing the Lexical Entry Process . . . 12

4 Beyond Lexical Entry 12

4.1 Testing Your Lexical Entries Before Exiting the Image . . . 13

4.2 After Exiting . . . 14

A Verb Complement Types 16

1 Introduction

1.1 Features

The Lexical Entry Procedure (LEP) has been designed to provide consistency, completeness, and speed of entry for new words. The procedure elicits relevant linguistic information from the user, computes dependencies between attributes, and prompts for morphologically related forms (offering a "guess" as to the correct form). The program then automatically creates a set of related dictionary entries, with as much structure-sharing among the entries as possible. Before the entries are actually entered in the database or written to a file, the user may inspect and edit any entries created.

1.2 Limitations

The LEP is a tool which relieves the user of some, but not all, of the burden of maintaining a lexicon. In its current version, it can only be used to add new lexical entries, and cannot be used to revise, delete, or display existing lexical entries. Furthermore, it does not directly access a lexicon: rather, it adds lexical entries to the Prolog database in a running image, and optionally copies them to a temporary file. The user must move the entries from the temporary file(s) created by the LEP to the appropriate lexicon.

What this means to you, as a user, is that you will need to become familiar with the LEP (as documented in this Guide), and you will also need to understand the tools for editing and deleting raw lexical entries. This in turn means that you will need to understand the structure and content of lexical entries in the form in which they are stored in a lexicon, for example (for the word *dog*):

```
:(dog,root:dog,[n:[11,singular],11:[ncount1]])
:(dogs,root:dog,[n:[11,plural]])
:(dog's,root:dog,[ns:[11,singular]])
:(dogs',root:dog,[ns:[11,plural]])
```

To help you with this task, we have included a number of sample lexical entries created by the LEP, and have included a brief section on what to do when you have completed the LEP.

2 Getting Started

2.1 How to access the LEP

The LEP may be run by itself, or may be called automatically whenever PUNDIT encounters an unknown word.

- Standalone

  Simply type the command **lep.** at the Prolog prompt in a PUNDIT image. You will be prompted for the word whose definition you wish to add.

- During text processing

  If you have set the PUNDIT switch **enter_new_word**[1] to **on**, the LEP will be invoked automatically whenever PUNDIT encounters an unknown word in its input. When this happens, you will be given the opportunity to respell the word, to add the word to the lexicon, or to abort processing. If you chose to add the word, the LEP will be invoked. After you have completed the LEP, your definitions will be added to the image, and PUNDIT will resume processing with the new definitions.

  The transcrript below illustrates the first two options: **respell** and **add**. The input is **Ticonderoga attaked**, where **attaked** is a misspelling, and **ticonderoga** is a proper name which is not in the lexicon. The actual prompts which come up during lexical entry will be discussed in detail shortly. For now, observe that in the illustration, PUNDIT has successfully parsed the input after the spelling error was corrected and **ticonderoga** was defined.

```
| ?- turn_on(enter_new_word).

yes
| ?- parse.

sentence: ticonderoga attaked.
>>

No Lexical Entries found for: ticonderoga

Choose one of the following items

   1. respell          2. add              3. abort

Lexical Entry option: add

          Defining the lexical entries for 'ticonderoga'.

Output to a file? [yes] :

          Entries will be saved in the file muck_lexicon.pl.15Nov1956

Root form [ticonderoga]:
Other spellings [none]:
Word classes: name

          Defining 'ticonderoga' as a proper noun

Singular possessive [ticonderoga's]:

The following lexical entries have been created:

:(ticonderoga,root:ticonderoga,[proper:[]])
```

[1] For more information on PUNDIT switches, please consult [Ball 88].

```
:(ticonderoga's,root:ticonderoga,[ns:[11,singular]])

Enter? :yes

No Lexical Entries found for: attaked

Choose one of the following items

   1. respell          2. add              3. abort

Lexical Entry option: respell

respell: attacked

continuing processing with respelled word(s) -- [attacked]
```

2.2 General conventions

2.2.1 Meta-responses

All LEP prompts accept meta-responses, which begin with the special character '@' and end with a period.

- **@help.** - ask for help.
- **@help(<ITEM>).** - ask for help on a menu item.
- **@quit.** - abandon the current definition.
- **@prolog(<PROLOG COMMAND>).** - execute Prolog command.

2.2.2 Defaults

Many of the prompts in the LEP offer defaults. In a menu, the default is marked with an asterisk; otherwise, the default is enclosed in square brackets. To accept a default, press the RETURN key. Otherwise, enter your response. In the following example, the user has overridden the default plural form for *goose*, but accepted the default singular possessive form.

```
Plural form [gooses]: geese
Singular possessive [goose's]:
```

2.2.3 Menus

Certain prompts require a response from a fixed list of choices; these choices are shown as a menu when you ask for help. There are two basic types of menus: those from which you can select only

one item, and those which allow you to select multiple items. The menu title will indicate which is the case.

If the menu requires a single item as a response, you may enter either the number of the item, or the name of the item. If the menu allows multiple items, you may enter the numbers or the names, separated by commas. In the following example, the user is selecting the word classes for the word *fool*, which is both a noun and a verb:

```
Word classes: @help.

Choose one or more of the following

   1. noun            2. name            3. verb
   4. adjective       5. adverb          6. determiner
   7. quantifier      8. preposition

Select: noun,verb
```

Note that the user could also have entered `1,3` instead.

3 Defining New Words

3.1 Before You Begin

The LEP assumes that you are adding **new** lexical entries, and it does not check to ensure the entries are not already in the image you are running. If you want to be able to test your entries by parsing with them in this image, you should first verify that the word you wish to enter is not already defined. Do this by using the procedure **edit_word**, giving your word as argument. If the word has already been defined, this procedure will display the existing lexical entries. If you intend to completely replace them, delete each one. If you are sufficiently proficient, you can use **edit_word** to make any necessary revisions instead of using the LEP (but there will be no external record of your revisions).

If your word has not already been defined, or if you are not concerned about creating duplicates, proceed with the LEP.

3.2 Initial Prompts

The first five prompts in the LEP are common to all lexical entries:

```
****************************** Lexical Entry ******************************


Word: fool

          Defining the lexical entries for 'fool'.

Output to a file? [yes] :

          Entries will be saved in the file muck_lexicon.pl.15Nov1956

Root form [fool]:
Other spellings [none]:
Word classes: noun,verb
```

- **Word**

  Enter the word which you wish to add. What you enter will serve as the default for the **Root form** prompt, but plays no other role at present. The LEP is designed entirely around root forms.

- **Output to a file?**

  Answer **yes** if you wish to save your definitions into a file in the current working directory (the name of the file is automatically generated). If you answer **no**, your definitions will be recorded in the image you are running, but there will be no external record. In short, your definitions will be lost when you exit the image.

- **Root form**

  The root is the most basic form of the word in the same grammatical category. If the word is a verb, the root is the infinitive form; if the word is a noun, it is the singular form. If the word you are defining is an abbreviation, enter the full form (e.g. for *lb*, enter *pound*). If the word is a variant spelling, enter what you consider to be the standard spelling. Otherwise, the root form is generally identical to the word itself.

  Special problems arise when you need to define the root form of an acronym (e.g. *unodir* for *unless otherwise directed*) or a root which contains hyphens (*anti-aircraft*), or an idiomatic phrase (*go sinker*). These are all treated as 'multi-word expressions', and this version of the LEP cannot handle them. You will need to consult with a PUNDIT expert to determine how to enter them (manually) into a lexicon.

- **Other spellings**

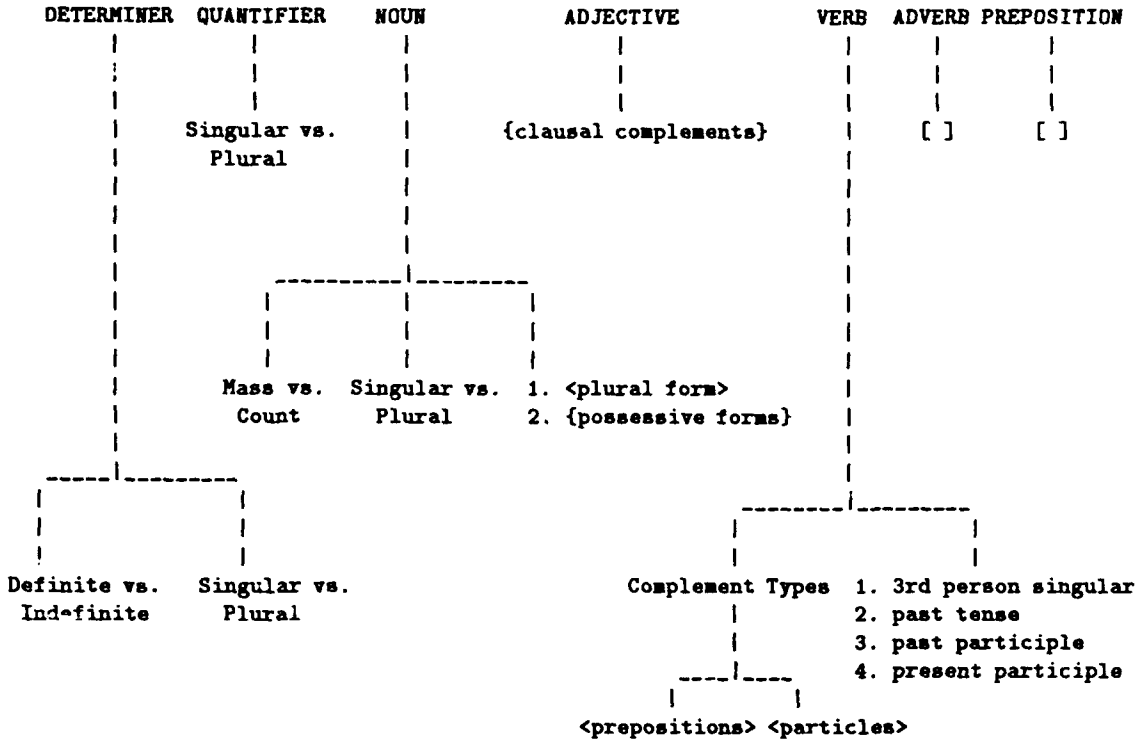
  Here, you can specify any variant spellings or abbreviations for the root, such as *sep,sept* for *september*, or *archaeology* for *archeology*.

  It is important to note that these are other forms of the **root**. Special problems arise when you wish to define an abbreviation for a non-root form, for example *clearng* as an abbreviation for *clearing*, or *lbs* for *pounds*. The LEP cannot handle these cases, and you will need to consult with a PUNDIT expert.

- **Word classes**

  Enter the part(s) of speech of the root. For example, if the root can be both a noun and a verb, enter **noun,verb**.

The next sections cover major word classes and their features in detail. The diagram below shows the features and morphological information which are collected for each word class. Items enclosed in { } are optional, while items enclosed in < > reflect information that the user may or may not be asked to provide, depending on previous choices.



3.3 Word Classes

3.3.1 Nouns

A noun is first classified as mass or count. If the noun is a count noun, the LEP prompts for number information and plural form (it is assumed that the root is singular). For both count and mass nouns, you will then be asked to specify the possessive forms. Sample definitions for *woman* (a count noun) and *mud* (a mass noun) are given below.

```
Word classes: noun

          Defining 'woman' as a noun

Count/Mass [count]:
Number [singular]:
Plural form [womans]: women
Singular possessive [woman's]:
Plural possessive [womans']: women's
```

```
The following lexical entries have been created:

:(woman,root:woman,[n:[11,singular],11:[ncount1]])
:(women,root:woman,[n:[11,plural]])
:(woman's,root:woman,[ns:[11,singular]])
:(women's,root:woman,[ns:[11,plural]])

Enter? :yes
```

Note that there is a lexical entry for each morphological variant. Each lexical entry consists of the citation form, followed by the root form, followed by a list of lexical classes and their attributes. This is data which is intended to be recognized by the parser. In the first entry, **n** indicates a noun, and **ncount** indicates a count noun. In the entries for *woman's* and *women's*, **ns** indicates possessive. The occurrence of **[11,...]** in the entries is a pointer to the basic feature **11:[ncount1]** in the first entry, where the root is classified as a count noun. You may find it enlightening to consult [Fitzpatrick 81] for a more detailed discussion of word classes such as **ncount1** in the context of a related system.

A sample definition for the mass noun *mud*:

```
Word classes: noun

          Defining 'mud' as a noun

Count/Mass [count]: mass
Singular possessive [mud's]:
```

3.3.2 Proper Nouns

You will be asked to specify the singular possessive form. Example:

```
Word: philadelphia

          Defining the lexical entries for 'philadelphia'.

Output to a file? [yes] :n
Root form [philadelphia]:
Other spellings [none]: philly,phila
Word classes: name

          Defining 'philadelphia' as a proper noun

Singular possessive [philadelphia's]:
```

Note that the name has been entered in lowercase letters. If it had been capitalized, the LEP input reader would have converted it to lowercase anyhow. This is because PUNDIT in general converts all input to lowercase letters, so it would be useless to allow the distinction to be made in the lexicon.

3.3.3 Verbs

Defining the characteristics of a verb is perhaps the most daunting of all lexical entry tasks. PUNDIT requires very detailed information about what types of complement a verb can take, and what prepositions and particles the verb requires. This information is necessary to get correct parses and avoid incorrect parses, but it is difficult to specify. Many of the distinctions amongst complement types may be obscure to the non-linguist, but they are all significant. You may find it useful to consult a dictionary such as Longman's for guidance. Since it requires a great deal of thought to determine the complement types of a verb, you may find it most efficient to work this out on paper, before using the LEP to record your decisions.

We have tried to simplify the task of specifying the complement types of a verb by offering three different menus: a menu for transitive uses of the verb, a menu for intransitive uses of the verb, and a menu for verbs which take clausal complements. The items in each menu are numbered, and you must choose one or more items by number.

Within each menu, you will be shown first the String Grammar name for the complement type, and then a short description. You may request help on any item by typing **@help(*NUMBER*).**, where ***NUMBER*** is the number of the item on the menu. The help messages give examples of verbs which take this complement type, and some criteria for making a decision. All of the complement types are discussed in more detail in [Linebarger 88], which is attached as an appendix to this guide.

After you have specified the complement types, you will be asked about tense and participial forms of the verb. A sample definition for the verb *think* is shown on the following pages.

```
Word classes: verb

          Defining 'think' as a verb

Takes a clausal complement? :yes

Choose clausal complement types, by number:
   1. tovo - infinitival complement, raising-to-subject
   2. eqtovo - infinitival complement, subject-controlled equi
   3. objtovo - noun phrase + infinitival complement, object-controlled equi
   4. ntovo - noun phrase + infinitival complement, raising-to-object
   5. thats - 'that'-clause
   6. assertion - 'that'-clause, but 'that' is optional
   7. nthats - noun phrase + 'that'-clause
   8. pnthats - prepositional phrase + 'that'-clause
   9. svo - tenseless clause with no complementizer
   10. clshould - 'that'-clause + subjunctive
   11. pnthatsvo - prepositional phrase + clshould
   12. snwh - indirect question
   13. nsnwh - noun phrase + indirect question
   14. sven - predicative 'small clause'
   15. sobjbe - small clause with subject
   16. dpsn - particle + clause
Clausal complement types: 5,6,15
```

```
Transitive? :y

Choose transitive complement types, by number:
   1. nstgo - noun phrase (simple transitive verb)
   2. npn - noun phrase + prepositional phrase
   3. pnn - prepositional phrase + noun phrase
   4. nn - double object dative
   5. na - noun phrase + adjective phrase
   6. dp2 - particle + noun phrase
   7. dp3 - noun phrase + particle
   8. dp2pn - particle + noun phrase + prepositional phrase
   9. dp3pn - noun phrase + particle + prepositional phrase
Transitive complement types: 6,7

Intransitive? :y

Choose intransitive complement types, by number:
   1. nullobj - no complement (simple intransitive verb)
   2. pn - prepositional phrase
   3. astg - adjective
   4. dstg - takes specific adverbs
   5. dp1 - particle
   6. dp1pn - particle + prepositional phrase
Intransitive complement types: 1,2

Predicative verb? [no] :
Prepositions for the PN complement: about
Particles for the DP2 complement: up
Particles for the DP3 complement: up

3rd person singular [thinks]:
Past tense [thinked]: thought
Past participle [thought]:
Present participle [thinking]:

The following lexical entries have been created:

:(think,root:think,[v:[12],tv:[12,plural],
        12:[objlist:[nullobj,thats,assertion,sobjbe,pn:[pval:[about]],
        dp2:[dpval:[up]],dp3:[dpval:[up]]]]])
:(thinks,root:think,[tv:[12,singular]])
:(thought,root:think,[tv:[12,past],ven:[14],
        14:[12,pobjlist:[assertion,objbe,thats,dp1:[dpval:[up]],
        p:[pval:[about]]]]])
:(thinking,root:think,[ving:[12]])

Enter? :yes
```

3.3.4 Adjectives

For adjectives, PUNDIT needs to know whether the adjective can take a clausal complement; if it can, which of four complement types. Help is available on each.

- **ordinary**

  Takes a *that*-clause as a right modifier, e.g. *I am glad that she won*, and the syntactic subject of the sentence is also the logical subject. The String Grammar name for this class of adjective is **asent3** (this is what you will see in the lexical entry created).

- **extraposition**

  It...adjective that..., e.g. *It is obvious that he is tired.*. The logical subject is the *that*-clause, which appears to have been extraposed to the right, leaving *it* behind. For adjectives of this type, there will often be acceptable versions with and without extraposition: *It is obvious that he is tired* and *That he is tired is obvious*. The String Grammar name for this class of adjective is **asent1**.

- **equi**

  Like the **ordinary** complement type, except that the clause is infinitival instead of a *that*-clause, and the subject of the sentence is also the understood subject of the infinitive. Example: *Bill is eager to please*, which means that *Bill* wants to do the pleasing. The internal name for this class of adjective is **aasp:[equi_adj]**.

- **raising**

  Like **extraposition**, except that the clause is infinitival, and the subject appears to have been raised out of the clause. Example: *She is certain to be re-elected.* The logical subject is the clause, with the syntactic subject put back into it: *That she will be re-elected is certain, It is certain that she will be re-elected.* The internal name for this class of adjective is **aasp:[raising_adj]**.

3.3.5 Adverbs

No special information is collected for adverbs.

3.3.6 Determiners

Determiners (articles) are classified according to definiteness and number: for example, *a* is indefinite and singular, while *the* is definite and both singular and plural. A sample definition for *another*, which is indefinite and singular:

```
Word classes: determiner

        Defining 'another' as a determiner

Definiteness: indefinite
Number [singular]:
```

3.3.7 Quantifiers

Quantifiers are classified according to number. A sample definition for *many* is given below:

```
Word classes: quantifier

        Defining 'many' as a quantifier

Number [singular]: plural
```

3.3.8 Prepositions

No special information is collected for prepositions.

3.4 Completing the Lexical Entry Process

After the LEP has collected all attributes of the root, it will display the lexical entries created, and you will be asked whether you wish to enter them. If at this point you realize that you have made an error in lexical entry, you can simply type **@quit.** and start over again - nothing has yet been saved to a file or added to the Prolog database.

If you answer **yes**, the entries will be recorded in the current image (and written to a file, if you so specified). You will then be given the opportunity to define more words.

If you answer **no**, you will be shown each of the entries created, one at a time, and you may choose to enter it, ignore it, or edit it. At this point you can also quit and start over again. Note that no action is taken until one of these choices is made for each of the entries. If you choose **ignore**, the entry will be thrown away. If you choose **edit**, you will enter the Prolog Structure Editor. This is a tool which requires some expertise, since you will be directly editing the raw lexical entry - consult [Riley 86] for more details. If you get into the Structure Editor by mistake, type **?** to see the options - one of these will be **a** to abort.

4 Beyond Lexical Entry

If you answered **yes** to **Output to a file?**, the lexical entries you created will now be in a temporary file in your directory, and you will need to eventually move them into the appropriate lexicon file. This must be done manually.

Before you move your lexical entries to a more permanent home, however, it is advisable to test, by devising test sentences and using PUNDIT to parse them. In the sections which follow, we describe how to test and correct errors at three different stages: before exiting your current image. after exiting the image but before moving the entries to a lexicon, and after moving the entries. The well-known rule applies here: the earlier you detect an error, the easier it is to fix it.

4.1 Testing Your Lexical Entries Before Exiting the Image

After you have completed the LEP, the lexical entries which you created have been stored in the Prolog database and are thus available to the PUNDIT parser. This is the most convenient stage at which to test the correctness and completeness of your entries.

Before you begin parsing, there are at least three PUNDIT switches you may wish to adjust: **text_mode**, **parse_tree** and **semantics**. Turn the **text_mode** switch **off** - this will enable you to obtain **all** the parses for your input. Turn the **parse_tree** switch **on** - so that you can see exactly how PUNDIT has analyzed your input. The **semantics** switch you may wish to turn **off**: if you have defined a verb, semantic analysis will not work properly until you have also defined the semantics rules for the verb; if you have defined a noun, semantic analysis will not work properly until you have defined the corresponding concept in the knowledge base.

As a result of testing, you may find that some aspect of your lexical entry was incorrect or incomplete. At this point there are several ways in which you can correct your error.

1. Exit the image and start all over again from scratch.
2. Use **edit_word** to delete the entire set of lexical entries from the Prolog database, and use the LEP to redefine them.
3. Use **edit_word** to revise the offending lexical entries. This option is not recommended for novices.

[Riley 86] explains how to use **edit_word**, but here is a simple example showing how to delete all the lexical entries for a given root.

```
| ?- edit_word(dog).

Editing a set of words with the same root
Word 1: :(dog,root:dog,[n:[11,singular],11:[ncount1]])
Word 2: :(dog's,root:dog,[ns:[11,singular]])
Word 3: :(dogs,root:dog,[n:[11,plural]])
Word 4: :(dogs',root:dog,[ns:[11,plural]])
Command: d4

Word number:4 is marked to be deleted   You may no longer edit it.
Editing a set of words with the same root
Word 1: :(dog,root:dog,[n:[11,singular],11:[ncount1]])
Word 2: :(dog's,root:dog,[ns:[11,singular]])
Word 3: :(dogs,root:dog,[n:[11,plural]])
Word 4: :(dogs',root:dog,[ns:[11,plural]])
Command: d3

Word number:3 is marked to be deleted.  You may no longer edit it.
Editing a set of words with the same root
Word 1: :(dog,root:dog,[n:[11,singular],11:[ncount1]])
Word 2: :(dog's,root:dog,[ns:[11,singular]])
```

```
Word 3: :(dogs,root:dog,[n:[11,plural]])
Word 4: :(dogs',root:dog,[ns:[11,plural]])
Command: d2

Word number:2 is marked to be deleted.  You may no longer edit it.
Editing a set of words with the same root
Word 1: :(dog,root:dog,[n:[11,singular],11:[ncount1]])
Word 2: :(dog's,root:dog,[ns:[11,singular]])
Word 3: :(dogs,root:dog,[n:[11,plural]])
Word 4: :(dogs',root:dog,[ns:[11,plural]])
Command: d1

Word number:1 is marked to be deleted.  You may no longer edit it.
Editing a set of words with the same root
Word 1: :(dog,root:dog,[n:[11,singular],11:[ncount1]])
Word 2: :(dog's,root:dog,[ns:[11,singular]])
Word 3: :(dogs,root:dog,[n:[11,plural]])
Word 4: :(dogs',root:dog,[ns:[11,plural]])
Command: t

Do you want to delete: :(dog,root:dog,[n:[11,singular],11:[ncount1]]).
Enter 'y' or 'n': y

Do you want to delete: :(dog's,root:dog,[ns:[11,singular]]).
Enter 'y' or 'n': y

Do you want to delete: :(dogs,root:dog,[n:[11,plural]]).
Enter 'y' or 'n': y

Do you want to delete: :(dogs',root:dog,[ns:[11,plural]]).
Enter 'y' or 'n': y
```

4.2 After Exiting

After you have exited from the image in which you were using the LEP, your entries will now reside only in the temporary file created by the LEP. At this point you may wish to test them (if you have not already done so), or you may wish to load them into an image for some other purpose.

To load your lexical entries from a file into an image, use the procedure **readIn**, whose argument is the name of the file containing your lexical entries. For example:

```
| ?- readIn('muck_lexicon.pl.18Nov1918').
```

If, however, some of your lexical entries are intended to **replace** definitions which are already in the image, you should first remove the old lexical entries. To do this, you can use **edit_word** and delete them one at a time. Or if you are not sure which words are already defined, you can remove **all** lexical entries from the image by using the procedure **rdb_remove(dict)** (see [Ball 88] for more information).

If you discover errors in your lexical entries at this stage (that is, while your entries are still in a temporary file created by the LEP), you can simply remove the temporary file, delete the entries from your image (using **edit_word**) and use the LEP to re-enter your definitions.

You may also discover errors in lexical entries after they have been moved to a lexicon file, and that file has been used to build an image. If the image is your own personal image, you can simply rebuild it after fixing the problem (and updating the lexicon). If the image is shared, you may need to follow whatever system administration procedures obtain at your site.

References

[Ball 88] Ball, Catherine N., Dowding, John, Lang, François-Michel, and Weir, Carl. *PUNDIT User's Guide.* Technical Report, Unisys Corporation, 1988.

[Fitzpatrick 81] Fitzpatrick, Eileen and Sager, Naomi. Appendix 3: The lexical subclasses of the LSP String Grammar. In Sager, Naomi (editor), *Natural Language Information Processing: A Computer Grammar of English and Its Applications*, pages 322–374. Addison-Wesley, Reading, Mass., 1981.

[Linebarger 88] Linebarger, Marcia C. *A Guide to Object Options in PUNDIT.* Technical Report, Unisys Corporation, 1988.

[Riley 86] Riley, Leslie and Dowding, John. *The Prolog Structure Editor.* LBS Technical Memo 29, Unisys Corporation, 1986.

PUNDIT
Lexical Entry Procedure
Mini-release Notes*

Version 1.2

December 14, 1988

Unisys Logic-Based Systems
Paoli Research Center
P.O. Box 517, Paoli, PA 19301

*This work has been supported by DARPA contract N00014-85-C-0012, administered by the Office of Naval Research.

The Lexical Entry Procedure (LEP) was extensively revised in Version 1.1 to improve ease of use by non-experts. In addition, minor bugs were corrected, and several obsolete prompts were removed.

In Version 1.2, a new lexicon display facility has been added, and the Lexical Entry Procedure has been extended to allow the entry of 'multi-word expressions'. These changes are described below.

1 Lexical Display

In order to make the PUNDIT lexicon accessible to the ordinary user, a lexical display facility has been developed. In this version of PUNDIT, the display is accessed by the SRE, and can also be run stand-alone, using the following Prolog commands:

- `lex_display_all.`

  Use this command to display the entire lexicon in the current image.

- `lex_display_all(WordClass).`

  Use this command to display the definitions of all words in a specified word class. For example, to display all the verbs in the current lexicon, type:

  `lex_display_all(verb).`

- `lex_display(Word).`

  Use this command to display the definition for a single word. If the word is a root form (for example, the infinitive of a verb or the singular form of a noun), all the variants of the root form will be displayed. Otherwise, only information for the particular word form will be displayed.

  For example, the display for the root form *attack*:

```
attack [noun,verb]
     n. count singular; pl. attacks; sing. poss. none; pl. poss. none
     v. present sing. attacks, pl. attack; past attacked
        past part. attacked; pres. part. attacking
          vt.
             nstgo - They attacked it
             npn - They attacked it [with] something
          vi.
             nullobj - It attacked
             pn - They attacked [on] something
```

  The display for the non-root form *attacked*:

```
attacked (root: attack) [verb]
     v. [past,past. part]
```

The displays show the definition of a word in what is intended to be a helpful and legible format, using essentially the same terminology as the Lexical Entry Procedure. For verb complement types (such as **nstgo** in the example above), templates are used to generate example sentences. Into these templates are inserted the past tense form of the verb (e.g. *attacked*) and any prepositions or particles which were specified. The latter appear in brackets in the examples (e.g. **[with]** was specified as the valid preposition for the **npn** complement of *attack*).

The current PUNDIT lexicon contains a number of lexical integrity problems: for example, some words have roots which are undefined; some verbs have invalid complement types; some entries have typing errors which make the entry unreadable to the display procedure. When one of these conditions is encountered, the display procedures print out a generic error message. The actual error can be pinpointed (if desired) by running the lexicon integrity checker, which has been separately developed and documented.

Please be aware that this version of the lexical displays is incomplete and deficient in several respects. We are only displaying information in the lexicon which the Lexical Entry Procedure understands, and it turns out that this is a subset of the actual information in the PUNDIT lexicon. In addition, we are currently unable to provide the correct treatment of words which have more than one root form, and we are not showing 'other spellings' of words. The next version of the displays will remedy these shortcomings.

It still remains possible to obtain a display of the raw physical database, if you wish. Two PUNDIT procedures exist which may be used for this purpose:

- **show_lex.**

  This procedure displays all the lexical entries in the Prolog recorded database, exactly in the form in which they are stored.

- **edit_word(Word).**

  This procedure, which is documented separately, can be used to edit the raw lexical entries in the Prolog recorded database for a specified word. But since it first displays all the lexical entries which have the same root as the specified word, it can be used as the 'physical' equivalent of the logical view offerred by **lex_display(Word)**.

2 Lexical Entry Procedure

The LEP has been enhanced to allow the entry of 'multi-word expressions' such as *anti-submarine rocket*. These are stored in the lexicon as single words joined by circumflexes, for example, as in the follwoing lexical entry:

```
:(asroc,root:anti^(-)^submarine^rocket,[n:[11,singular],11:[ncount1]]).
```

To enter such expressions using the LEP, simply enter them in the form in which they would appear in text, e.g.:

```
Word:  anti-submarine rocket
```

The LEP will transform this into the form in which it must be stored in the lexicon.

A Guide to Object Options in PUNDIT*

Marcia Linebarger

August 10, 1988

*This work has been supported by DARPA contract N00014-85-C-0012, administered by the Office of Naval Research. APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED.

Contents

1 Introduction 1

1.1 Handling of Passive in the Lexicon 1

1.2 The ISR 1

1.3 On pvals and dpvals 2

2 Object Options 2

2.1 NULLOBJ 2

2.2 NSTGO 2

2.3 PN 3

2.4 NPN 4

2.5 PNN 4

2.6 OBJBE 5

2.7 EQTOVO 5

2.8 TOVO 6

2.9 NTOVO 6

2.10 OBJTOVO 7

2.11 THATS 8

2.12 ASSERTION 8

2.13 PNTHATS 8

2.14 SVO 9

2.15 C1SHOULD 9

2.16 PNTHATSVO 10

2.17 SNWH 10

2.18 NSNWH 10

2.19 NTHATS 10

2.20 SVEN 10

2.21 NN 11

2.22 SOBJBE 12

2.23 NA 12

2.24 ASTG 13

2.25 DSTG 13

2.26 DP1 13

2.27 DP2 14
2.28 DP3 14
2.29 DP1PN 14
2.30 DP2PN 15
2.31 DP3PN 15
2.32 DPSN 15

1 Introduction

This document describes the current object options of the grammar, with the corresponding passobj (passive object) options and ISRs (Intermediate Syntactic Representations – see below), and with some very limited annotations on their structural quirks, semantics, raison d'être, and so forth. The numbering of object options below is the same as that in the Lexical Entry Procedure, and these notes are intended for use during entry of new lexical items. Object options which are restricted to one or two verbs (such as BE_AUX, VENO, and VO, associated with the auxiliaries *be*, *have*, and modals) are not included in this list, because we assume that most verbs with these subcategorizations have already been entered in the lexicon. Such object types may be assigned to a new verb by choosing **Other** in the Lexical Entry Procedure menu.

1.1 Handling of Passive in the Lexicon

The parse tree built by PUNDIT represents surface structure; transformations such as passivization and wh-movement are not 'undone' at this level. Thus verbs must be subcategorized for the objects they take in both active and passive. (Note on terminology: objects of the verb in its active form are called **object**; the list of a verb's objects in the lexicon is called the **objlist**. Similarly, passive objects are called **passobj**, and the list of a verb's passive objects in the lexicon is called the **pobjlist**. Note the systematic ambiguity of the word 'object'.) Because the correlation between an active and a passive object is predictable, the Lexical Entry Procedure automatically computes the passobj on the basis of the active objects selected. Verbs which do not passivize receive no pobjlist whatsoever in the lexicon; they should not be subcategorized for NULLOBJ in the passive. The *by*-phrase, if present, is parsed as a sentence adjunct rather than a passobj. Note that although some active object options (e.g., NULLOBJ) are never associated with a corresponding passive object, since they never passivize, others may or may not be; since the Lexical Entry Procedure automatically computes the corresponding passobj for any object type which passivizes, it is up to the user to edit out of the lexical entry any unacceptable passobj.

1.2 The ISR

Although the parse tree represents surface structure, the ISR is a somewhat more abstract level of syntactic representation, which, like the 'deep structure' of transformational grammar, provides a more transparent representation of argument structure. For example, the surface subject of the passive is represented in the ISR as the object of the verb. As in many current syntactic theories, the subject position of a passive ISR remains unfilled (in PUNDIT, it is filled with the dummy element **passive**), and it is the function of semantic rules to determine whether an element in a *by*-phrase may fill the semantic role which would be assigned to the subject. Thus, at least for the object, active and passive sentences can be interpreted by the same semantic mapping rules. In some cases, the ISRs of passive sentences diverge significantly from the surface structure in order to bring about this parallelism between active and passive; for example, the ISR for a pseudopassive such as *The patient was operated on* reconstructs the prepositional phrase. Thus the surface parse tree provides the bare preposition *on* as object of the verb, while the ISR provides the prepositional phrase *on the patient* as object.

The ISR also fleshes out the argument structure of constructions such as equi and raising, as seen in connection with object types EQTOVO, TOVO, and OBJTOVO below; and it regularizes the surface

order of object types which differ from one another only in the order of their components (such as NPN and PNN, or DP2 and DP3).

Because there are such divergences between the ISR and the surface parse, and because the ISR plays an important role as the interface between syntax and semantics, the ISRs associated with each object type and its passivized counterpart are given below. For ease of exposition, only the prettyprinted ISR is displayed.

1.3 On pvals and dpvals

Object types containing prepositions can be subcategorized for particular prepositions, via **pval** sublists in the lexicon; object types containing particles can be subcategorized for specific particles via **dpval** lists in the lexicon. The Lexical Entry Procedure queries the user to create these lists where appropriate.

2 Object Options

2.1 NULLOBJ

A verb which takes no complement at all is subcategorized for NULLOBJ. Example: *The pump failed*, which receives the following ISR:

```
OPS:  past
VERB: fail
SUBJ: the pump (sing)
```

Such verbs do not passivize, hence there is no corresponding passobj.

2.2 NSTGO

This is the simple transitive verb option, a noun phrase non-predicative direct object. Example: *She repaired the sac*, which receives the following ISR. The direct object receives the semlabel **obj**. (Semlabels are applied to elements in the ISR to label those grammatical functions which play a role in semantic rules. In the prettyprinted ISRs, the semlabels of all postverbal elements appear in capital letters, e.g. **SUBJ**: in the example below.)

```
OPS:  past
VERB: repair
SUBJ: pro: she (sing)
OBJ:  the sac (sing)
```

The passobj counterpart of NSTGO is NULLOBJ, as in *The sac was repaired (by her)*. The *by*-phrase is parsed as a sentence adjunct; this is not evident in the ISR below because the ISR (for reasons having to do with the functioning of the semantic interpreter) fails to indicate whether a prepositional phrase occurs as a sentence adjunct or a verb object.

```
OPS:   past
VERB:  repair
SUBJ:  passive
OBJ:   the sac (sing)
PP:    by
       pro: her (sing)
```

Note that the surface subject is represented as the object in the ISR. The subject position of the ISR is filled with the dummy element **passive**.

2.3 PN

This is a prepositional phrase object. Example: *They operated on him*:

```
OPS:   past
VERB:  operate
SUBJ:  pro: they (pl)
PP:    on
       pro: him (sing)
```

Corresponding passobj: isolated preposition. Example: *He was operated on*; in the ISR, the prepositional phrase is reconstructed:

```
OPS:   past
VERB:  operate
SUBJ:  passive
PP:    on
       pro:   he (sing)
```

When do we want PN to be analysed as an object option rather than a sentence adjunct (SA)? As far as I can tell, the following are the most relevant cases in which the PN object is subcategorized for in this system:

(a) The verb is unacceptable with NULLOBJ, and PN will suffice. E.g., **He told* (ignoring elliptical reading). But *He told of great adventures.*

(b) The VERB + PN has an idiomatic meaning (or just feels like a unit): *the surgeon operates on the patient* and *the surgeon operates on the table* represent, under their most plausible readings, the PN object and SA attachments respectively. Similarly: *Bill turned into the side street* (SA expressing where he turned) vs. *Bill turned into an orangutang* (PN object).

The possibility of a pseudopassive doesn't seem to be a motivating factor: *sleep* in our lexicon isn't subcategorized for *in* or *on*, etc., yet you can say *That bed was slept in by George Washington* or *This floor has been slept on by countless fatigued partygoers.* If a verb with PN object can passivize at all, as above, its passobj will be a P (at the moment this passobj is not listed under very many verbs in the lexicon.) Thus it is currently an unsolved problem how to treat pseudopassives

corresponding to active sentences in which the PN is in SA as in the *sle_p* example above: we don't really want to allow P as an SA option generally. So another possibility would be to allow PN object (with no subcategorization for specific lexical items) more freely, automatically generating the PN object possibility for ANY verb which allows ps udopassive. The cost of this is that we lose the way of structurally representing differences such as that between, e.g., *operate on the table* and *operate on the patient.*

2.4 NPN

and

2.5 PNN

NPN consists of an NSTGO followed by a PN, as in *They returned the disk drive to the factory*:

```
OPS:   past
VERB:  return
SUBJ:  pro:   they (pl)
OBJ:   the disk^drive (sing)
PP:    to
       the factory (sing)
```

See above for discussion of when to include the PN in object rather than SA. Another criterion: is there a corresponding PNN object? PNN is the BNF node associated with NPN which has undergone a shifting of the NP, constrained by various stylistic factors such as heaviness. It's one of the unpleasant facets of the grammar we use that this extraposition gets expressed as a different BNF node. Subcategorization for PNN follows redundantly from subcategorization for NPN, since the acceptability of PNN depends not on the verb but on the NP itself. (Compare *He presented to us an enormous chocolate cake iced with yellow daffodils* vs. the much less pleasing *He presented to us a cake.*)

Note that a sequence of NP + PN need not be parsed as NPN; for example, *I found Louise in a state of euphoria* should probably be classed as a SOBJBE (see below), given related sentences such as *I found Louise euphoric, I found Louise a changed woman.* The PN here is predicated of Louise rather than simply being an argument of *find.* In contrast, the PN in *I found Louise on the fourth try* seems more like an SA describing the circumstances of the event of finding Louise, certainly not a predication stating that Louise was *on the fourth try.*

The passobi counterpart of NPN/PNN is PN, as in *The disk drive was returned to the factory*:

```
OPS:   past
VERB:  return
SUBJ:  passive
OBJ:   the disk^drive (sing)
PP:    to
       the factory (sing)
```

(Compare **The ... tory was returned the disk drive to*: no pseudopassive is possible here except with idiomatic expressions such as *He was given a talking to.*)

2.6 OBJBE

OBJBE, the object type associated with *be* as a main verb, is subcategorized for by verbs other than *be*. OBJBE expands to an NP, an adjective phrase, or a PP; not every verb allows all these expansions, as indicated by bvals in the lexicon. (The Lexical Entry Procedure does not currently solicit bvals.) Examples: *The pump appears inoperative*:

```
OPS:   present
VERB:  appear
SUBJ:  the pump (sing)
ADJ:   inoperative
```

and *She became a field engineer*:

```
OPS:   past
VERB:  became
SUBJ:  pro:  she (sing)
PREDN: a field^engineer (sing)
```

These verbs don't passivize at all, so they have no passobj counterpart (and hence no pobjlist is created for them by the Lexical Entry Procedure.)

Thus an NP following the verb can be analysed either as an NSTGO (*He photographed the President's advisor*) or as an OBJBE (*He became the President's advisor*). This enforces the well-known fact that predicative verbs do not passivize: *The best cars are made by the Japanese* (active form: nstgo) vs. **The best cooks are made by Italians* (active form: objbe).

2.7 EQTOVO

An example of EQTOVO is *The fe wants to repair the disk drive*. EQTOVO corresponds to what is traditionally known as an infinitival complement with subject controlled equi; the subject of the matrix verb is understood to be also the subject of the infinitive. This is made explicit in the ISR, where the matrix subject is copied into the infinitive; the ID variables for the two NPS are identical (a fact which is obscured below because the ISR prettyprinter does not display variables):

```
OPS:   present
VERB:  want
SUBJ:  the field^engineer (sing)
OBJ:   OPS:   untensed
       VERB:  repair
       SUBJ:  the field^engineer (sing)
       OBJ:   the disk^drive (sing)
```

There is no passobj, as these structures do not passivize.

2.8 TOVO

An example of TOVO is *The pump seems to be failing.* The TOVO object corresponds to what is traditionally known as raising; the matrix subject is analysed as an argument of the infinitive, but not of the matrix verb, which has the infinitive as its sole argument. This is made explicit in the ISR, where the reconstructed infinitival clause is the subject:

```
OPS:   present
VERB:  seem
SUBJ:  OPS:   untensed,prog
       VERB:  fail
       SUBJ:  the pump (sing)
```

As for passobj, raising verbs don't passivize, so there is no pobjlist.

As noted above, these two object types EQTOVO and TOVO differ in their argument structure, and hence in their selection properties, differences which are made explicit in the ISR. In the EQTOVO (equi) case, the phonologically null subject of the infinitive undergoes selection with the matrix verb as well as with the verb in the infinitive. That is, *the fe* is really the subject of both *want* and *repair* in *The fe wants to repair the disk drive.* One can run afoul of selection restrictions between this noun and *either* verb: *The number 12 wants _ to be divisible by 3,* and *The cat wants _ to be divisible by 3* are both anomalous, due to violations of selection between the matrix subject and the matrix and embedded predicates, respectively.

For the bare TOVO case, the matrix subject is semantically just the subject of the lower verb; that is, the matrix verb is really a one-place predicate with a clause as its argument. (Thus the ISR subject of *The pump seems to be failing* is not *the pump* but *the pump to be failing.*) There's no selection between the surface NP subject (*the pump*) and this matrix verb (*seem*): whatever can be subject of the infinitival verb V can also be subject of *seem to V...D.* Sager refers to these as aspectual verbs. They include: *seem, appear, start, tend, continue, come* (as in *It came to rotate,* NOT as in *I come to bury Caesar, not to praise him.* The latter is a purposive TOVO in SA.)

To summarize: with EQTOVO, the matrix subject is an argument of the matrix verb and also of the verb in the infinitive; with TOVO, the matrix subject is an argument only of the lower (infinitival) verb. (The two types correspond to equi and raising, respectively.)

In Sager's grammer, these two categories are conflated. Some existing lexical entries therefore require updating, since this distinction was introduced after PUNDIT's lexicon was established.

2.9 NTOVO

Like OBJTOVO (see below), NTOVO is associated with surface sequences of the form 'NP to VP' following the matrix verb; it corresponds to what is sometimes called 'exceptional case marking (ECM)'. An example of NTOVO is *The factory expects the fe to repair the sac:*

```
OPS:   present
VERB:  expect
SUBJ:  the factory (sing)
OBJ:   OPS:   untensed
```

```
VERB:  repair
SUBJ:  the field^engineer (sing)
OBJ:   the sac (sing)
```

Thus *the field engineeer* is the subject of the clause but is not a direct object of the matrix verb; the factory does not expect the fe, but rather it expects the proposition expressed by the infinitive. (A consequence of this is that pleonastic elements such as *there* may occur in subject position of NTOVO: *I expect there to be unlimited champagne.*)

The passobj counterpart of NTOVO is TOVO, as in *The fe is expected to repair the sac*; the ISR rule associated with TOVO will automatically reconstruct the infinitive *the fe to repair the sac*:

```
OPS:   present
VERB:  expect
SUBJ:  passive
OBJ:   OPS:   untensed
       VERB:  repair
       SUBJ:  the field^engineer (sing)
       OBJ:   the sac (sing)
```

2.10 OBJTOVO

OBJTOVO corresponds to object controlled equi; in *The factory told the fe to repair the pump, the fe* is an argument (indirect object?) of the matrix verb and subject of the infinitive:

```
OPS:   past
VERB:  tell
SUBJ:  the factory (sing)
D_OBJ: the field^engineer (sing)
OBJ:   OPS:   untensed
       VERB:  repair
       SUBJ:  the field^engineer (sing)
       OBJ:   the pump (sing)
```

The semlabel d_obj (dative object, formerly known as inner_obj) is used here to capture the parallelism with *The factory told the fe the truth.*

The passobj counterpart is EQTOVO. The ISR rules associated with EQTOVO reconstruct infinitive as above for *The fe was told to repair the sac*:

```
OPS:   past
VERB:  tell
SUBJ:  passive
D_OBJ: the field^engineer (sing)
OBJ:   OPS:   untensed
       VERB:  repair
       SUBJ:  the field^engineer (sing)
       OBJ:   the sac (sing)
```

Major differences between NTOVO, OBJTOVO: in NTOVO, the subject of the infinitive is an argument ONLY of the lower verb. The entire infinitival clause is itself the argument of the matrix verb. There are no selection restrictions between, e.g., *believe* and *the table* in *I believed the table to be quite attractive.* In OBJTOVO, on the other hand, the noun phrase between the matrix verb and the infinitive is an argument of BOTH matrix and embedded predicates, as demonstrated by the anomaly of *I persuaded the table to seat 6* (violates selectional constraints on *persuade*) and *I persuaded the man to be divisible by 2* (violates selectional constraints on *divisible*). also, NTOVO but not OBJTOVO allows *there* as subject: *I expect there to be a diplomat at the party* (**I persuaded there to be a diplomat at the party*).

PUNDIT does not currently handle the rare cases of subject-controlled equi in verb complements of the form 'NP to VP', as in *Mary promised Louise to arrive on time.* This form of control is largely restricted to the single verb *promise.*

2.11 THATS

and

2.12 ASSERTION

THATS and ASSERTION are both tensed clauses, with and without the complementizer *that*, as in *The fe said that the disk drive was inoperative*:

```
OPS:   past
VERB:  say
SUBJ:  the field^engineer (sing)
OBJ:   OPS:   past
       VERB:  be
       SUBJ:  the disk^drive (sing)
       ADJ:   inoperative
```

Verbs subcategorized for THATS and ASSERTION are automatically subcategorized for these same objects in the passive, given the possibility of pleonastic subjects, as in *It is said that whales are highly intelligent.* Work remains to be done to constrain these cases in the grammar. General note on passobjs with verbs taking clausal objects (ASSERTION, THATS, PNTHATS, SVO, C1SHOULD, SNWH, NSNWH, NTHATS): in Sager, passives with *it* subject (*It was reported that the disk failed*) are not treated as having a clausal passobj. Rather, the clause goes into `rv` at the string level. However, it seems to me that these verbs should all be subcategorized for clausal passobj.

2.13 PNTHATS

This is a PN followed by THATS, as in *The fe reported to the factory that the sac had failed*:

```
OPS:   past
VERB:  report
SUBJ:  the field^engineer (sing)
```

```
PP:    to
       the factory (sing)
OBJ:   OPS:   past,perf
       VERB:  fail
       SUBJ:  the sac (sing)
```

These objects are further subcategorized for pvals, like all PN-containing objects. Not every VERB + PP + CLAUSE structure involves a PNTHATS; for example, *this proves with some certainty that the world is round* should be analyzed as a THATS with preceding PN in SA, while *this proved to everyone that the theory was wrong* should be treated as PNTHATS with PN in OBJECT.

The passobj counterparts are PN and PNTHATS, as in *It was revealed to us yesterday that the company had gone bankrupt* (PNTHATS as passobj), or *That Smith was the culprit was announced to the entire assembly* (PN as passobj).

2.14 SVO

SVO is a tenseless clause; it differs from C1SHOULD (see below) in that (1) SVO never has the complementizer *that*, (2) a pronoun subject of SVO is accusative. Example: *She saw them replace the pump*:

```
OPS:   past
VERB:  saw
SUBJ:  pro:   she (sing)
OBJ:   OPS:   untensed
       VERB:  replace
       SUBJ:  pro:   them (pl)
       OBJ:   the pump (sing)
```

Passivization is not acceptable out of SVO, cf. **They were seen replace the pump.*

2.15 C1SHOULD

This consists of the complementizer *that* followed by SVO, as in *He suggested that it be replaced*:

```
OPS:   past
VERB:  suggest
SUBJ:  pro:   he (sing)
OBJ:   OPS:   untensed
       VERB:  replace
       SUBJ:  passive
       OBJ:   pro:   it (sing)
```

Passobj counterparts: C1SHOULD, as in *It was suggested that we leave early*; and probably NULLOBJ. (My intuitions are unclear on NULLOBJ as passobj here.)

A pronoun subject of C1SHOULD is nominative. The current BNF rule for C1SHOULD requires *that*, but should be generalized to account for *I suggest we leave*.

2.16 PNTHATSVO

This consists of PN followed by C1SHOULD, as in *I suggested to Bill that he write up his investigations.* Pvals are elicited by the Lexical Entry Procedure. Passobj counterparts are PN and PNTHATSVO.

2.17 SNWH

Not currently implemented. This is an indirect question, an embedded clause beginning with a wh-word. Example: *I know who borrowed the car, She wondered whether it would snow.* Passobj counterparts are SNWH and NULLOBJ, as in *It was finally revealed who stole the car,* or *What he was really up to that day was revealed months later at the investigation.*

2.18 NSNWH

Not currently implemented. This is an NP object followed by indirect question, as in *He asked us whether it would snow.* Passobj counterparts: SNWH, NULLOBJ.

2.19 NTHATS

This is an NP followed by a THATS, as in *She told the factory that the sac was inoperative*:

```
OPS:   past
VERB:  tell
SUBJ:  pro:   she (sing)
D_OBJ: the factory (sing)
OBJ:   OPS:   past
       VERB:  be
       SUBJ:  the sac (sing)
       ADJ:   inoperative
```

Note that the NP object is marked as a dative object (semlabel d_obj, formerly inner_obj). This is because of the parallelism with dative constructions like *He told the factory the truth.*

Passobj counterpart: THATS. The semlabelling of this construction in passive is currently being refined in order to distinguish between cases like *He was told that the pump was inoperative*, where the subject should be marked as d_obj; and *It was said that the pump was defective*, where expletive *it* should not be represented in the argument structure at all.

2.20 SVEN

This is a predicative small clause, as in *He had the sac repaired quickly*:

```
OPS:   past
VERB:  have
SUBJ:  pro:   he (sing)
OBJ:   OPS:   untensed
       VERB:  repair
       SUBJ:  passive
       OBJ:   the sac (sing)
       ADV:   quickly
```

This sentence is ambiguous between SVEN and NSTGO analyses of the object: the NSTGO reading can be paraphrased *He had the sac which had been repaired quickly*, while the SVEN reading can be paraphrased *He caused the sac to be repaired quickly*. In the latter case, no one need be in possession of the sac. This difference is clearer still in *She found the book missing*. Clearly, *book* is not itself an argument of *find*, since the book was not found; what was found (out) was the proposition *the book is missing*. There's a lot of variation here, though: sometimes the subject of the small clause under *find* also seems to be an argument of the verb, especially in the passive (*The car was found parked on Elm Street*). Other verbs are clearer: *They reported the car stolen* doesn't mean that they reported the car, nor does *He had the stairs fixed* mean that he had the stairs. Probably one should split hairs and use two different BNF nodes corresponding to the NTOVO vs. OBJTOVO (exceptional case marking vs. object-controlled equi) distinction.

Passobj counterpart: VENPASS, as in *The gear teeth were found stripped and corroded.* SVEN doesn't always passivize, as above. (ISR rule is still under development for this passobj.)

2.21 NN

NN is the double object dative, as in *The factory found her a new pump* or *They told her the result*:

```
OPS:   past
VERB:  tell
SUBJ:  pro:   they (pl)
D_OBJ: pro:   her (sing)
OBJ:   the result (sing)
```

Note that the indirect object is semlabelled `d_obj`.

Passobj counterpart is NSTGO, as in *She was told the result*:

```
OPS:   past
VERB:  tell
SUBJ:  passive
D_OBJ: pro:   she (sing)
OBJ:   the result (sing)
```

Note that NP + NP sequences need not be parsed as NN. *I gave Ruth a good answer* contains NN, but *I consider Ruth a good dancer* is SOBJBE (below).

Many but not all NNs have counterparts with the *to-* or *for-* dative; thus *give books to Louise* alternates with *give Louise books*. However, in some cases only the prepositional form is found

(compare the meaning of *I got my degree for my parents (not for myself)* with that of *I got my parents my degree*); in other cases, we find only NN, as in *The book cost Mary five dollars.* The two constructions (NN and prepositional datives) have different semantic properties, so we do not want to attempt to represent them identically in the ISR.

2.22 SOBJBE

This is another small clause, consisting of subject followed by OBJBE (nstg, astg, or pn), as in *I consider him a genius* or *They consider it inoperative*:

```
OPS:    present
VERB:   consider
SUBJ:   pro:    they (pl)
OBJ:    OPS:    untensed
        VERB:   be
        SUBJ:   pro:    it (sing)
        ADJ:    inoperative
```

Sager has further subcategorization for nstg or astg or pn (or dstg, not included here) via **bvals** in the lexicon, since some verbs do not allow all OBJBE options: cf. *That made her angry, That made her the reigning monarch, *That made her in a state of rage.* PUNDIT's Lexical Entry Procedure does not currently elicit **bvals**.

The passobj counterpart is OBJBE, as in *He is considered a genius by his associates* or *It is considered inoperative*:

```
OPS:    present
VERB:   consider
SUBJ:   passive
OBJ:    OPS:    untensed
        VERB:   be
        SUBJ:   pro:    it (sing)
        ADJ:    inoperative
```

2.23 NA

This is a sequence of NP followed by an adjective phrase, as in *She painted the barn red* or *they stripped the gears bare*:

```
OPS:    past
VERB:   strip
SUBJ:   pro:    they (pl)
OBJ:    the gear (pl)
RES_CL: OPS:    untensed
        VERB:   be
        SUBJ:   the gear (pl)
        ADJ:    bare
```

The NA object type differs from SOBJBE in several respects. First, in NA the NP is an argument of the verb; if one paints the barn red, one has definitely painted the barn, whereas to have found the book missing is not to have found the book, and to believe the problem insoluble is not to believe the problem. Furthermore, the predication relationship between the adjective phrase and the NP is interpreted as a result in the case of NA. Finally, there is sometimes idiosyncratic selection between verb and adjective in NA, but not in SOBJBE. Thus *We sanded it smooth* sounds fine, but *We sanded it ugly* sounds odd, even if the ugliness is interpreted as resulting from the sanding.

The passobj counterpart is ASTG, as in *The house was painted red* or *It was stripped bare*:

```
OPS:    past
VERB:   strip
SUBJ:   passive
OBJ:    pro:    it (sing)
RES_CL:OPS:     untensed
       VERB:    be
       SUBJ:    pro:    it (sing)
       ADJ:     bare
```

2.24 ASTG

Example: *It went bad*:

```
OPS:    past
VERB:   go
SUBJ:   pro:    it (sing)
ADJ:    bad
```

Verbs with the ASTG object select for particular adjectives, as in *He went mad* (vs. the anomalous *He went sane*); and do not subcategorize for other OBJBE options (**He went a madman*). But it seems semi-semantic: *He turned blue/green/mean/sour/serious* but **He turned old/happy*. Thus it might not be possible to subcategorize for specific lexical items.

No passive.

2.25 DSTG

This is also quite rare. Certain verbs subcategorize for specific adverbs (*He means well* vs. **He means warmly*, or *She did beautifully* vs. **She did quietly*). No passive.

2.26 DP1

This is the simplest verb-particle combination, as in *He showed off*, *We lined up* (vs. **He showed out*, **We lined over*), or *Engine jacks over*:

```
OPS:   present
VERB:  jack
SUBJ:  engine (sing)
PTCL:  over
```

No passive.

2.27 DP2

DP2 is a particle followed by an NP, as in *He ran up the bill.* In contrast, *He ran up the hill* in its normal interpretation is NOT a DP2, but is rather a PN object. One test: only particles can occur to the right of the noun: *He ran the bill up* vs. **He ran the hill up*, to cite a classic example. Another test: only a PN can be topicalized, since it's a constituent: *Up the* HILL *he ran* vs. **Up the* BILL *he ran.* Another example: *They blew up the ship*:

```
OPS:   past
VERB:  blow
SUBJ:  pro:   they (pl)
PTCL:  up
OBJ:   the ship (sing)
```

Passobj counterpart is the particle, DP1, as in *A huge bill was run up that evening* or *The ship was blown up*:

```
OPS:   past
VERB:  blow
SUBJ:  passive
OBJ:   the ship (sing)
PTCL:  up
```

2.28 DP3

DP3 is just the permuted version of DP2, where the particle follows the noun phrase. Same passobj as DP2; order regularized in ISR. Since there are no transformations in PUNDIT, such alternations as that between DP2 and DP3 must be handled lexically.

2.29 DP1PN

This is a particle followed by a PP: *She moved in on him, They found out about it, The factory should have followed up on it*:

```
OPS:   past,shall,perf
VERB:  follow
SUBJ:  the factory (sing)
PTCL:  up
PP:    on
       pro:   it (sing)
```

Passobj counterpart is DP1P, when it passivizes, as in *The announcement was led up to by a series of remarks about the company's financial difficulties*(?), or *It should have been followed up on*:

```
OPS:   past,shall,perf
VERB:  follow
SUBJ:  passive
PP:    on
       pro:   it (sing)
PTCL:  up
```

2.30 DP2PN

DP2PN is a DP2 (particle + NP) followed by a PN, as in *He mixed up the apples with the pears.*

Passobj counterpart: DP1PN, as in *The apples were mixed up with the pears.* (Not, for example, **The pears were mixed up the apples with.*)

2.31 DP3PN

This is a DP3 (NP + particle) followed by a PN, as in *mix the apples up with the pears.* Passobj counterpart is also DP1PN.

2.32 DPSN

DPSN is a particle followed by a clause, as in *She found out where it was hidden, He pointed out that it was noon already, They often make out to be villains*, or *She found out that it was inoperative*:

```
OPS:   past
VERB:  find
SUBJ:  pro:   she (sing)
OBJ:   OPS:   past
       VERB:  be
       SUBJ:  pro:   it (sing)
       ADJ:   inoperative
PTCL:  out
```

Passobj counterparts are DPSN, as in *It was pointed out frequently that the plan could not succeed*, and DP1 *Where it was hidden was never really found out.* Both sound a little marginal, but might occur.

A User's Guide to the Selection Module[†]

François-Michel Lang

LBS Technical Memo No. 68
Paoli Research Center, Unisys
P. O. Box 517, Paoli, PA 19301

1. Introduction

This guide is intended to introduce the PUNDIT community to the selectional component, and to answer any questions that users may have about its use and operation. Improvements and suggestions are most welcome.

2. Rationale

The purpose of this module is to collect empirically observed word-level selectional patterns from data, and to support generalisation of these patterns to semantic class patterns. These patterns are classified into valid and invalid patterns, and stored in a pattern database.

3. Basics

The selectional component is invoked from the BNF grammar by two restrictions: `vso_selection` and `np_selection`.

- `vso_selection` is called from the CENTER and ASSERT_FRAG nodes, to check selection in assertions, questions, and fragments.
- `np_selection` is called from the NSTG node, to check selection in LNR nodes.

The BNF rules in question are the following:

```
:- center ::=
 ((({dquest2}, assertion, {w_endmark} -> assertion),{vso_selection}) xor
   (({dquest1}, question, {w_endmark} -> question), {vso_selection}) xor
              ((fragment, {w_endmark} -> fragment), {vso_selection})) xor
 (compound -> compound), {vso_selection}.

:- assert_frag ::=
   ((assertion, internal_punct -> assertion) xor
    (fragment, internal_punct -> fragment)), {vso_selection}.

:- nstg ::=
   ({d_endmark},
      (((lnr -> lnr),{np_selection});
        (lpror -> lpror);
        (nsvingo -> nsvingo)));
   ({d_gap},nullwh -> nullwh).
```

[†]This work has been supported in part by DARPA under contract N00014-85-C-0012, administered by the Office of Naval Research (APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED), in part by National Science Foundation contract DCR-85-02205, as well as by Independent R&D funding from System Development Corporation, now part of Unisys Corporation.

These two restrictions then examine the ISR for those nodes to check selectional patterns in the assertion, question, fragment, or noun phrase. At the time these restrictions are called, the ISRs are expected to be instantiated, simplified, fully assembled and lambda-free. If the restrictions encounter an ISR which is not in simplified operator-operator form, a very visible warning message will be issued to the user by the "soop checker". Assuming the ISR is well formed, each of the two restrictions then calls a definite-clause grammar (DCG) to analyze the ISR.

4. At the Top Level

After typing in a sentence, the user will be asked to enter a unique sentence ID if that sentence has not yet been recorded in the current corpus of sentences.

The parser will then parse away, and when a complete LNR or sentential node has been assembled, the ISR for that node will be passed to the DCG, and the questions will begin.

5. The Queries

In the course of examining the ISR, the selection mechanism will ask certain questions about the validity of lexical co-occurrence patterns. Some typical questions (with some sample answers in italics) are

```
Is this        <svo> pattern good: field^engineer repair sac ---------> y
Is this     <qpos/n> pattern good: <NUMBER> sac ----------------------> y
Is this       <n/pp> pattern good: loss of sac ----------------------> y
Is this        <n/n> pattern good: sac failure ----------------------> y
Is this      <adj/n> pattern good: fine particle---------------------> y
```

The question contains two important parts:

- the type of pattern (e.g., svo, qpos/n, n/pp, n/n, adj/n)
- the specific lexical items which form that pattern (In certain cases, special atoms such as <NUMBER> will appear in the pattern instead of actual lexical items. These special atoms are discussed in more detail below).

6. The Pattern Types

The types of patterns are listed in Figure 1 below (this list is subject to change), with examples for any non-obvious patterns. The names of the patterns will eventually change, since currently the slash ("/") is overloaded, denoting conjunction, modification, and siblinghood.

7. The Responses

When prompted with such a lexical pattern, the user has several possible responses:

(1) "y": (YES) Signals a globally good pattern. Answer with **y** when the pattern is semantically acceptable, consistent with the domain, and plays the intended role in the sentence (i.e., leads to a correct parse).

Figure 1: Types of Selectional Patterns

PATTERN	EXPLANATION and EXAMPLES of GOOD PATTERNS
adj/n	An adjective (either attributive or predicate) modifying a noun EX: *FINE metal PARTICLES found in filter.*
adj/pp	A PP functioning as adjective complement EX: *Oil is DARK IN APPEARANCE.*
adv/adj	An adverb modifying an adjective EX: *Sac is COMPLETELY INOPERATIVE.*
adv/p	An adverb modifying a preposition EX: *Pressure is SIGNIFICANTLY OVER the limit.*
conj/adj	Conjoined adjectives EX: *Loss of pressure was SUDDEN and UNEXPECTED.*
conj/n	Conjoined nouns EX: *Loss of PRESSURE and TEMPERATURE.*
conj/v*	Conjoined main verbs of 2 sentences Ex: *The sac BROKE, and the fe REPAIRED it.*
n/adj	A noun modifying an adjective EXS: *FACTORY INSTALLED, CRYSTAL CLEAR*
n/adv	An adverb in an NSTG_FRAG EX: *Sac FAILURE YESTERDAY.*
n/n	A compound noun EX: *Loss of OIL PRESSURE.*
n/pp	A PP modifying a noun EX: *EROSION OF IMPELLOR is evident*
n/predn	A subject and predicate noun EX: *Alarm CAPABILITY is a NECESSITY.*
nq	An NQ consists of a noun followed by a Q; EX: *See FIGURE 3.*
nq/n	An NQ modifying a noun EX: *the FIGURE 3 STATISTICS*
qn	A QN consists of a Q followed by a noun. EXS: *10 DAY, 2 INCH*

qn/n — A QN modifying a noun
EXS: *a FIVE ALARM FIRE, a 500 PAGE BOOK*

qpos/n — A quantifier modifying a noun
EX: *The fe repaired 5 SACS.*

svo — Subject verb object
EX: *The FE REPAIRED the SAC.*

v/adv — An adverb modifying a verb
EX: *The sac FAILED SUDDENLY.*

v/pp — A PP modifying a verb
EX: *Metal particles were DISCOVERED IN OIL FILTER.*

v/qn — An NQ modifying a verb
No known examples in the sac domain
A muck example is *Ship CLEARING 60 DEGREES*

*Patterns marked with an asterisk are not currently presented to the user because they contain no significant selectional information.

---

(2) "n": (NO) Signals a globally bad pattern. Answer with ***n*** when the pattern is semantically unacceptable, or not consistent with the domain.

(3) "s": (SUCCEED) Signals a locally good pattern. Answer with ***s*** in either of two situations. One is when the pattern is semantically unacceptable, and not consistent with the domain, but happens to be part of the right parse for the sentence. These are rare, and are usually caused by "phrasal attributes" (the "***stiff neck*** phenomenon"). A discussion of this troublesome phenomenon is included in the next section. A example from the sac domain is found in the sentence ***Oil pressure dropped below 30 psig***, which generates the PP pattern [***drop, below, psig***], which is anomalous, but, at least in this parse, correct. The other case in which one should answer ***s*** is when one wishes to defer judgement about a pattern whose validity or acceptability in the domain is in doubt. If the user is not willing to categorically state that the pattern is anomalous, but, on the other hand, is not convinced of its validity, ***s*** is the correct response.

(4) "f": (FAIL) Signals a locally bad pattern. Answer with ***f*** when the pattern may be semantically acceptable and consistent with the domain, but happens to be part of a wrong parse. Example: The sentence ***Loss of oil pressure*** might generate the pattern [***loss, of, oil***], which may be semantically valid, but is not part of the right parse.

(5) "a": (ABORT) Abort parsing this sentence. More on this option later.

(6) "b": (BREAK) Enter a break level. Has effect of typing ***b*** to the debugger or calling the goal ***break*** in Prolog (which is, of course exactly what this does).

(7) "e": (EXPLAIN/EXPAND/EXAMPLES) Ask for an explanation of the pattern and additional examples of such patterns. (This feature has not yet been fully implemented.)

(8) "h" (HELP) Ask for a summary of possible answers.

(9) "?" Same as *h*

8. The Phrasal Attribute Problem

As mentioned above, "Locally good" patterns are used to deal with phrasal attributes. An example of this phenomenon taken from a medical domain is the noun phrase ***stiff neck***: The semantic class of the head noun of this NP, ***neck***, is something like BODY-PART, but the semantic class of the full NP ***stiff neck*** is not BODY-PART, but rather SYMPTOM or AILMENT. This discrepancy between the semantic classes of the full NP and of its head noun presents a difficulty in making a decision about the acceptability of patterns generated. For example, in parsing the sentence ***Patient has stiff neck***, the system would present to the user the SVO pattern [***patient, have, neck***]. Note that this is indeed the correct syntactic parse (in fact, probably the only one), but we do not want to assert for posterity that the SVO pattern [***patient, have, neck***] is semantically acceptable in a medical sublanguage.

This is perhaps a subtle point, but not everything that is true in a sublanguage can be said in that sublanguage. The sentence ***Patient has stiff neck*** is a case in point: Although it is certainly true that the patient has a neck, nobody would ever (bother to) say so because the proposition is completely uninformative. Indeed, it is one of the characteristics of a sublanguage that certain (true) information is presupposed, and never explicitly stated.

In short, the parse is good, but the pattern [***patient, have, neck***] is bad. We do not want to say the pattern is good, but saying it is bad will fail the parse, and that is not a desirable result either. Hence the appropriate response to the query about this pattern would be to tag it as "locally good", which is a sort of compromise implemented in order to allow the parse to succeed, but without entering the pattern in question into the (global) pattern database.

Our method of dealing with this phenomenon is admittedly not satisfactory. However, pending a fuller semantic treatment of NPs which allows such distinctions to be made, it at least permits the correct parse to be obtained without creating obviously bad patterns.

For an example of the phrasal-attribute phenomenon from the SAC domain, consider the sentence ***Start air pressure dropped below 30 psig***, which generates the PP pattern ***drop below psig***. The problematic NP here is ***30 psig***: the semantic class of the head noun ***psig*** is UNIT-OF-MEASUREMENT, yet we would not say that the full NP ***30 psig*** is a UNIT-OF-MEASUREMENT. ***30 psig*** is instead an entity of the class LEVEL or perhaps THRESHOLD. The problem is that in evaluating the pattern ***drop below psig***, we would realise that pressure can drop below a certain level or below a certain threshold, but it cannot drop below a unit of measurement. The solution is to tag this pattern as locally good.

9. Special Atoms Appearing in Patterns

There are a number of special atoms which can appear in a pattern. Using such an atom in a pattern usually serves one of two purposes:

- To generalise a pattern immediately. For example, the qpos/n pattern [*5, sac*] should have the same selection as [*6, sac*], so we generalize both these patterns to [*<NUMBER>, sac*] on the fly. The generalisation applies to numbers, dates, times, and other entities whose specific value or instantiation is irrelevant for selectional purposes. All that is relevant for such an entity is simply that it is in fact, e.g., a date. Typically, these special atoms are productive forms recognised by PUNDIT's shapes component.

- To serve as a placeholder for an entity whose internal structure is irrelevant to selection (e.g., <CLAUSE>), or whose referent is not inferrable from the ISR (e.g., <SOMEBODY/THING>, <WH>).

The special atoms are the following:

(1) <NUMBER>: stands for a number.
Ex: ***5 sacs failed*** will generate the qpos/n pattern [***<NUMBER>, sac***].

(2) <TIME>: stands for a time.
Ex: ***the fe repaired the sac at 1150T*** will generate the v/pp pattern [***repair, at, <TIME>***].

(3) <DATE>: stands for a date.
Ex: ***The fe repaired the sac on 12/25-2359*** will generate the v/pp pattern [***repair, on, <DATE>***].

(4) <PART_NO>: stands for a part number.
Ex: ***The fe repaired 123-456*** will generate the svo pattern [***field^engineer, repair, <PART_NO>***].

(5) <CLAUSE>: stands for a clause.
Ex: ***The fe reported that the sac failed*** will generate the svo pattern [***field^engineer, report, <CLAUSE>***].

(6) <SOMEBODY/THING>: stands for a passive or elided constituent.
Exs: The sentences ***Repaired the sac*** and ***The sac was repaired*** will both generate the svo pattern [***<SOMEBODY/THING>, repair, sac***].

(7) <NULL>: stands for a null object.
Ex: ***Sac failed*** will generate the svo pattern [***sac, fail, <NULL>***] (Do not confuse this with ***nulln*** below).

(8) <WH>: stands for a wh-word.
Ex: ***Who repaired the sac?*** will generate the svo pattern [***<WH>, repair, sac***].

(9) nulln: stands for a null head noun.
Ex: ***2 broke*** (as in ***The fe installed 4 sacs, and 2 broke***) will generate the qpos/n pattern [***<NUMBER>, nulln***] and the svo pattern [***nulln, break, <NULL>***]. (This last pattern is hardly perspicuous, and will need treatment by some mechanism designed to handle referential information.)

The use of some of these special atoms (specifically, <SOMEBODY/THING>, <NULL>, <WH>, and nulln) is not always intuitive, and is likely to change in the near future.

10. Generalisation to Class-Level Patterns

After answering the word-level query with either ***y*** or ***n***, the user will then be asked to form a generalisation of that pattern based on the information in the domain isa hierarchy, provided,

of course, that there is a hierarchy.[1]

The interface for this section is still in flux, but the current state of affairs is as follows: After answering that a given pattern is good (or bad), the user is shown all possible generalizations for each word in the pattern appearing in the domain hierarchy. For example, when generalising the SVO pattern [*miller, sight, kynda*], the user would be shown the output in Figure 2 below.

Each line of the display in Figure 2 shows a path from the concept in question up to the children of the root concept. The user is then asked to choose which, if any, of the concepts are correct generalisations of the lexical item.

Figure 2: Generalizations Based on the Domain Hierarchy

The SVO pattern is [*miller, sight, kynda*]

```
****************************** GENERALIZING MILLER

These are the possible generalizations for MILLER:

miller knox us_platform platform physical_object

miller knox us_platform platform platform_group physical_object

miller knox frigate ship surface_platform platform physical_object

miller knox frigate ship surface_platform platform platform_group physical_object

****************************** GENERALIZING KYNDA

These are the possible generalizations for KYNDA:

kynda ur_platform platform physical_object

kynda ur_platform platform platform_group physical_object

kynda cruiser ship surface_platform platform physical_object

************************************************************************

Please enter the generalizations for MILLER (or type "!help." for help).
Generalizations: >>
```

---

[1]The selection mechanism expects the hierarchy to be encoded in clauses of the form `isa(Sub,Super)` and `semantic_type(Sub,Super)`.

The intention of the generalizations is that, given the good (bad) pattern *P*, which contains the word *SUB*, *SUB* generalizes to *SUPER* in the pattern *P* iff for every concept *C* such that *C* isa* *SUPER* (where isa* is the transitive closure of isa), the pattern *Q*, which is *P* with *C* substituted for *SUB*, is also a good (bad) pattern.

The help message (printed in response to typing "help." to the prompt

```
Generalizations: >>
```

in Figure 2) is quite informative, and makes available a number of useful options. The help message reads as follows:

```
Type your choices separated by commas, and terminated with a period.
List format is not necessary.
Type "[]" if you do not want to generalize at all.

Type "!break" to enter a break level.
Type "!abort" to abort parsing this sentence.
Type "!subs" to see the sub concepts of a concept.
Type "!supers" to see the super concepts of a concept.
Type "!help" to generate this message.
```

By invoking the commands described in the help message, the user can

- enter a break level (just like at the word-level prompt)
- abort out of parsing (more on this option below),
- ask to see all the immediate sub concepts of a given concept
- ask to see all the immediate super concepts of a given concept
- generate the help message.

After invoking either the `!subs` or `!supers` options, the user is prompted for the name of the concept whose descendants or ancestors are to be displayed.

11. Files

There are two files which the selection module can write to (or create): `SELECTIONAL_PATTERNS.pl` and `USER_CORPUS.pl`. Both files are in the current working directory. Since selection expects to be able to write to those files in the current working directory, users should ensure that they have write permission to the current working directory in order to run with selection on and save output to a file.[2]

The `SELECTIONAL_PATTERNS.pl` file is used to store the patterns that the user has been queried about. The file contains lines of the form

```
:-record_pattern(of,
```

[2]It is possible to change the current working directory (while running under Emacs) by typing *Esc-x cd* followed by the desired directory. The current directory can also be changed directly through Prolog by executing the goal

```
| ?- unix(cd(DIR)).
```

where `DIR` in the goal is an atom corresponding to a valid directory. Note: Changing the current working directory via Prolog does not allow specifying a path beginning with "~" (tilde), but the *Esc-x cd* method does.

```
    bad_selectional_pattern(n/pp,[loss,of,second],user(2))).
:-record_pattern(and,
    good_selectional_pattern(conj/n,[sac,and,disk],user(f4))).
:-record_pattern(fail,
    good_selectional_pattern(svo,[sac,fail,'<NULL>'],user(f4))).
```

Once such a file has been created, one need only compile it, and the patterns will be loaded in. The `USER_CORPUS.pl` file is used to store the sentences that the user has parsed with selection on. The file contains lines of the form

```
:-recordz(casreps,id([s2],[the,sac,failed,'.']),_660).
:-recordz(casreps,id([s4],[the,sac,and,the,disk,failed,'.']),_393).
```

The sentences stored in this form are used for parsing sentences in batchmode and for the `test_pundit` procedure.

The following are the other selection-related files in the stable system, and the contents of each file:

(1) `selection_dcg.pl`: The DCG to parse the ISR.

(2) `selection_query.pl`: The query and generalization mechanism.

(3) `selection_restr.pl`: The two selection restrictions (`vso_selection` and `lnr_selection`).

(4) `selection_tools.pl`: The selection switches, and facilities for inspecting, deleting, and editing patterns.

(5) `selection_top_level.pl`: The interface between selection and the PUNDIT top level, and various predicates to inspect and erase sentences recorded in a corpus.

(6) `selection_utilities.pl`: Miscellaneous utility predicates used by the selectional component.

(7) `xxx_selection_db.pl`: Domain-specific files (*xxx denotes the* domain) containing the selectional patterns originally stored in the `SELECTIONAL_PATTERNS.pl` file. Creating the `xxx_selection_db.pl` file must be done manually by gathering all selectional patterns collected in the `SELECTIONAL_PATTERNS.pl` file, verifying their correctness, and then putting the resulting set of patterns in the `xxx_selection_db.pl` file in the appropriate PUNDIT directory.

12. Selection Switches

There are several switches which can be used to control the behavior of the selection component. These switches have not yet been incorporated into the top-level pundit switches mechanism (as they should be), so the way to use and control these switches is likely to change. However, this is how they currently work.

To check the current setting of the selection switches, type the goal `sswitches`. Every switch has a default setting, indicated below by "(*)", and one or more associated predicates to control the setting of the switch. The switches currently supported in the selection mechanism are:

(1) **unknown_selection**: Controls the action of the program upon encountering an unknown selectional pattern. The possible settings are
- query: unknown patterns generate query to user (*)

- succeed: unknown patterns automatically succeed
- fail: unknown patterns automatically fail

To enable querying of unknown patterns, type the goal **squery**.
To allow unknown patterns to succeed, type the goal **ssucceed**.
To force unknown patterns to fail, type the goal **sfail**.

(2) **fileIO**: Controls whether or not the selectional patterns generated are written to the file `SELECTIONAL_PATTERNS.pl` in addition to recording them in the recorded DB. The possible settings are:
- ON : patterns are output to file (*)
- OFF : patterns are not output to file

To turn on **fileIO**, type the goal `fileIO(on)`.
To turn off **fileIO**, type the goal `fileIO(off)`.

Turning fileIO off will cause a dramatic increase in the real-time (but not the cpu time) efficiency of selection, but then the patterns won't be saved to a file.

(3) **pattern_trace**: Controls the printing of trace messages detailing selectional patterns generated and found in the pattern database. Possible settings are:
- ON : tracing messages are printed for every pattern generated, showing the lexical pattern found in the ISR, the class pattern generated from the lexical pattern, and any good or bad pattern found in the database which match either the lexical or class pattern generated.
- OFF : no tracing messages are printed (*)

To enable the pattern trace, type the goal `pattern_trace(on)`.
To disable the pattern trace, type the goal `pattern_trace(off)`.

(4) **lnr_trace**: Controls the printing of trace messages showing the ISRs of LNRs being fed to the DCG. Possible settings are:
- ON : the ISRs are printed
- OFF: the ISRs are not printed (*)

To enable the LNR trace, type the goal `isr_trace(lnr,on)`.
To disable the LNR trace, type the goal `isr_trace(lnr,off)`.

(5) **svo_trace**: Controls the printing of trace messages showing the ISRs of SVOs being fed to the DCG. Possible settings are:
- ON : the ISRs are printed
- OFF: the ISRs are not printed (*)

To enable the SVO trace, type the goal `isr_trace(svo,on)`.
To disable the SVO trace, type the goal `isr_trace(svo,off)`.
To enable both LNR and SVO traces, type the goal `isr_trace(on)`.
To disable both LNR and SVO traces, type the goal `isr_trace(off)`.

The three tracing switches (3), (4) and (5) are used to help debug the selection mechanism, and are probably of little interest to anyone else, as far as I can imagine, so most people will probably want to leave them set at their defaults!

13. Inspecting, Deleting, and Editing Selectional Patterns

After parsing merrily along for a while, the user might want to see what selectional patterns have been recorded, and possibly to delete or change some incorrect ones. A large number of predicates have been provided for inspecting, deleting and editing selectional patterns. All these predicates have to call a massive *setof*, so if there are a great many selectional patterns recorded, they can take a while.

13.1. Inspecting Patterns

To see all the selectional patterns currently recorded, type the goal `check_selection`.

To see all the selectional patterns currently recorded which contain the word *W*, type the go. `check_selection(word,W)`.

To see all the selectional patterns currently recorded which were generated by sentence *S*, type the goal `check_selection(sent,S)`.

To see all the selectional patterns currently recorded of a given type *T* (e.g., svo, n/adj, v/pp, etc.), type the goal `check_selection(exact_type,T)`.

If one is unsure of the exact type of the pattern one is looking for, all is not lost. The goal `check_selection(general_type,T)` will show all the selectional patterns currently recorded which contain *T* as one of its components (i.e., one of the constituents on either side of the "/" in the name of the pattern). For example, if one wants to see a pattern including a *qn*, but one is not sure if the specific pattern *is*, *say*, a *n/qn* or a qn/n, typing the goal `check_selection(general_type,qn)` will show *all* patterns of any type which includes a *qn*.

There is at present no mechanism for examining all selectional patterns containing a word of a given semantic class.

13.2. Deleting Patterns

There are variations of all of the predicates described above which can be used to delete selectional patterns. Instead of typing `check_selection` (with either 0 or 2 args) one should type `erase_selection` (with either 0 r 2 args).

`erase_selection` (0 arguments) will erase all selectional patterns. Period. There is no prompting, and no confirmation, so be careful! Note that this can also be done using `rdb_remove`.

However, using instead one of the following goals

```
erase_selection(word,W).
erase_selection(exact_type,T).
erase_selection(general_type,T).
erase_selection(sent,S).
```

will present all the relevant patterns (e.g., all patterns containing the word *W*), and ask which ones, if any, to delete. As is the case with examining patterns, there is at present no mechanism

for deleting all selectional patterns containing a word of a given semantic class.

For example, if one wanted to delete patterns containing the word ***break***, one wou
`erase_selection(word,break)`. A possible result would be

```
These are the patterns containing "break" which are currently sto
1:      (BAD)       <v/pp>     [break,off,attack]
2:      (BAD)       <v/pp>     [break,off,situation]
3:      (GOOD)       <svo>     [<SOMEBODY/THING>,break,engagemen
4:      (GOOD)       <svo>     [<SOMEBODY/THING>,break,process]
Enter your choices to delete ("h." = help) ==>
```

The patterns are be numbered for reference. The prompt is explicit about what to ans
in case of doubt, typing "h." as an answer will generate the following even-more-exp
message:

```
Please enter one of the following:
  -- the numbers of the patterns you want to delete (e.g., "1, 2,        23")
  -- "all" to delete all patterns
  -- "none" to delete none
followed by a period.
```

In order to delete some, but not all the patterns, one need not type the numbers in a
interface here is extremely flexible. The numbers can be typed in separated by comma
by hyphens if one wants to delete a range of patterns. For example, to delete patter
6, 7, 8, 9, 10, and 23, one can just type "1, 2, 5-10, 23."

Another nice thing about these predicates is that they will not stonewall: For examp
user ask for patterns of type *T*, and although *T* is valid pattern type, there don't hap
any patterns of that type recorded, (e.g., if the user asks to see SVO patterns, but th
SVO patterns recorded), a message will be printed, warning that

```
There are no patterns of type svo currently recorded.
```

However, if *T* is not a valid pattern type at all, the user will be told that *T* is not a
tern type, and a message will be shown presenting the valid pattern types.

One last aspect of deleting patterns involves the "abort" answer to the selectiona
query, which should be used if the user enters an incorrect answer.

After answering *a* (for ***abort***), the following will happen: First, the user is given a
undo the command to abort. If the user does indeed want to abort parsing, all patterns gen-
erated by the current sentence will be presented (in the format shown above), and the
be given a chance to delete any, all, or none of them. The user will then be given a
erase the current sentence itself (in case the sentence itself was incorrectly entered), and finally,
the parsing will abort, and Prolog will return to the top-level prompt.

Note that the `erase_selection` family of predicates will only affect the state of selection in
the current Prolog session. It is the user's responsibility to make appropriate modifications to
any files (e.g., `SELECTIONAL_PATTERNS.pl`) which contain the selectional data.

13.3. Editing Patterns

There are variations of all of the predicates described above which can be used to edit selectional patterns. This mechanism uses the Prolog Structure Editor. Instead of typing `check_selection` or `erase_selection`, just type `edit_selection`. (This predicate exists only in a 2-argument version).

Calling

```
edit_selection(word,W).
edit_selection(exact_type,T).
edit_selection(general_type,T).
edit_selection(sent,S).
```

will present all the relevant patterns (e.g., all patterns containing the word *W*), and ask which ones, if any, to edit. The acceptable responses are the same as those for deleting patterns. Once the user has selected which patterns to edit, the mechanism will then invoke the Prolog Structure Editor on each pattern selected, and modify the selection DB accordingly.

It is again the user's responsibility to make appropriate modifications to any files (e.g., `SELECTIONAL_PATTERNS.pl`) which contain the selectional data.

14. Future plans

There are a number of specific areas in which the selection module needs to be modified, some of which have been noted previously:

- Improving the treatment of anaphoric and elided elements (such as <SOMEBODY/THING> and nulln) in selectional patterns to allow the propagation of attributes deduced by selection.
- Extending the explanation facility (the "r" option) to the word level selection prompt.
- Optimising the matching of class-level patterns to word-level patterns. Several approaches have been considered. One suggestion has been to allow uninstantiated logic variables to be part of patterns. This solution has been partly implemented, but certain problems have not been solved concerning how to index on patterns containing variables. The approach of compiling the *isa* hierarchy directly into Prolog unit clauses has also been tried. The result was a noticeable gain in execution time, but at the cost of compiling in a large file containing approximately 1000 unit clauses. Another technique to be considered is the use of narrowing or feature intersection (à la LOGIN).
- Automatic generalization or success for certain specific patterns. For example, any part-whole relation in a noun/noun pattern such as [*submarine,hull*] should be allowed to automatically succeed.
- New names should be used for the patterns because the slash "/" has been overloaded in the name of patterns, since it denotes conjunction, modification, and siblinghood.
- The ability to examine, delete, and edit all patterns of a certain semantic class should be added.
- The selection switches should be be incorporated into the top-level switches mechanism.

PUNDIT'S SYNTACTIC COMPONENT
DESCRIPTION OF COVERAGE

Lynette Hirschman

1. Introduction to String Grammar

This document will provide an overview of PUNDIT's approach to syntax, based on string grammar (Z. Harris 1968, N. Sager 1981). Following the overview, the coverage of PUNDIT is sketched followed by a subsection providing some information on debugging tools and strategies for debugging PUNDIT's grammar.

PUNDIT's Restriction Grammar is an adaptation of Sager's well-documented *Linguistic String Grammar*. Since our approach has been driven by the need to cover constructions in the particular texts we were dealing with, we have added constructions to PUNDIT as needed. This means that by and large the PUNDIT's coverage of standard English is a subset of the grammar given in Sager's book, although PUNDIT contains additional constructions not documented in the book, such as an extensive treatment of sentential fragments. Also, over the years, we have deviated from Sager's string grammar treatment. Some deviations are minor (a more uniform treatment of modal verbs) and some are major (the meta-rule treatment of conjunction and of wh-clauses). These will be discussed in later subsections. For a general overview of string grammar, Sager's book remains the best reference work. Many of the "missing" constructions in PUNDIT could be readily added by consulting Sager's treatment.

One of the major extensions to Sager's system is the use of regularization rules with each production in the grammar. These rules describe, in a form of lambda calculus notation, how to combine the daughters of a given node into a regularised operator-operand notation that normalizes syntactic relations and makes explicit many of the gapped elements. The output of the regularization is called the *Intermediate Syntactic Representation* or *ISR*. It is the ISR that is passed on to semantics and selection, since it is far more regular than the surface syntactic analysis. However, the ISR, its rules and its mechanism will be discussed separately; this section will focus exclusively on syntactic coverage.

String Grammar Rules and Restrictions

String grammar is written as context-free rewrite rules, in the form of BNF definitions, augmented by *restrictions*, which are constraints on the well-formedness of the (partial) parse tree. As the BNF definitions are applied in string grammar, a partial parse tree is built up, corresponding to the definitions applied. The restrictions are interspersed with BNF definition expansion, and check to see that the parse tree constructed so far is well-formed. Restrictions are used to check for things such as agreement (subject-verb, determiner-adjective-noun), object subcategorization (so that verbs take only objects that they are subcategorized for), and positional constraints. These are the well-formedness restrictions, which fire on node completion. In addition, there are a number of optimization restrictions, that check to see if the pre-conditions for a particular construction hold. These are "disqualify" restrictions, which fire before node construction begins. A (simplified) grammar rule might be that **assertion** constructs a subject, followed by a verb, followed by execution of the **w_agree** well-formedness restriction, followed by construction of the object.

```
assertion ::= subject, verb, {w_agree}, object.
```

Here we see the well-formedness ("w_") agreement restriction, firing on completion of the verb node, to check subject-verb agreement.

Strings and LX... Nodes

String gra...mar distinguishes two types of constructs: *Head/adjunct* (endocentric) constructions and ...gs (exocentric) constructs. An endocentric construction has a *head* flanked by left modifiers ...d right modifiers; the behavior of an endocentric construction is governed by its head; that ...noun phrase is noun-like in its behavior; an adjective phrase is "adjective-like", etc. The ...ntric constructions are called lxr constructions in string grammar, where x stands for th... flanked by its left (l) and right (r) modifiers, which may be empty. The modifier nodes ...alled lx and rx respectively (where X is the head of the construction). Below are liste... of the important lxr constructions in string grammar; terminal nodes (lexical classes) a...ated by an asterisk. Note that the basic lexical classes (nouns, adjectives, verb and verb ...iples) have associated lxr constructions.

```
        ::= ln, nvar, rn.        % nvar = noun or variant (e.g., gerund)
        ::= la, *adj, ra.        % adj = adjective
        ::= lq, *q,rq.           % q = quantity word, e.g., "two"
        ::= lv, *v, rv.          % v = infinitive form of verb
        ::= lv, *tv, rv.         % tv = tensed verb
        ::= lv, *ving, rv.       % ving = present participle of verb
        ::= lv, *ven, rv.        % ven = past participle of verb
```

For verbs, t... and right modifiers (lv, rv) consist of slots for adverbials, e.g., **not** or **quickly**. Fo...tives, the left modifier la consists of an adverb slot for adverbs such as **verb**; the right mo...t ra consists of a list of options, including prepositional phrase (*suspicious of something*)...ntial complement (*certain that they left*) and adverbials. For nouns, the left modifier, ln ...of slots for the determiner (tpos = *the position*), quantity (qpos), adjectives (apos) and n...odifiers (npos). The right noun modifier (rn) consists of a list of options for prepositional ... relative clause, adjective, appositive, etc.

```
lv ::= ...null.          % *dstg = adverb slot
rv ::= ...dstg; null.    % pn = preposition + noun = prepositional phrase
la ::= ...null.
ra ::=
                         % that + sentence

ln ::=  qpos, apos, npos.
rn ::=
        ...lause;
```

The oth... important construction in string grammar is the *string*. A string is an exocentric construct... , that is, a construction whose behavior differs from that of its constituents. For example, an assertion cannot be considered "verb-like" or "subject-like" in its syntactic properties. A string is made up of two or more obligatory elements; the elements of a string are *obligatory*, except for the sa or sentence adjunct elements. Examples of important string constructions are:

```
assertion ::= sa, subject, sa, ltvr, sa, object, sa.
                                % sa = sentential adjunct
pn ::= *p, nstgo.               % prepositional phrase
qn ::= lqr, *n.                 % e.g., "two foot"
```

```
vingo ::= lvingr, sa, object. % e.g., "reading a bock"
vo ::= lvr, sa, object.       % e.g. "read a book"
venpass ::= lvenr, sa, passobj. % passive, "seen by me"
tovo ::= to, lvr, sa, object. % e.g, "to read a book"
```

Strings include basic constructions such as assertion, question, and imperative. The prepositional phrase (pn) is also a string, since its behaviour is neither that of a preposition *nor* that of a noun (in fact, it is often adverbial).

The philosophy of string grammar is to include a slot for each element; the realization of that element may be the empty string if the element is optional (e.g., adjuncts), or if it has been "zeroed" (reduced to zero or null) for some other reason (e.g., gapping). The advantage to this approach is that the skeletal parse tree is very regular. For example, an assertion always contains nodes for subject, verb, and object, separated by sentence adjunct slots. However, many of those nodes may be empty (including the object can be realized as **nullobj** for an intransitive verb). The adherence to this philosophy reduces the number of grammar rules and makes for efficient top-down parsing, but also makes for bushy trees with many empty nodes.

Object Options in String Grammar

One large group of strings is the class of objects. String grammar handles auxiliaries as instances of verb + complex object. This gives a very regular, recursive structure to the object node in string grammar. At the top level, we have the tensed verb, followed by an object. If the tensed verb is a modal, its object will be **vo** -- an infinitive followed by object, e.g., *I may read the book.* If the tensed verb is *be*, the object may be the participial object **vingo**, e.g., *I am reading a book*, etc. This means that objects carry a great deal of information, and may often contain the "meaning bearing" verb, where there are auxiliaries, as in *It may have to be reviewed*, where *review* is the meaning bearing verb, embedded in successive objects, as follows:

```
assertion
|
subject     sa itvr sa object
|           |       |
IT          MAY     vo
                    |
                    lvr   sa  object
                    |         |
                    HAVE      tovo
                              |
                              to verb sa object
                              |  |      |
                              TO BE     venpass
                                        |
                                        lvenr sa passobj
                                        |        |
                                        REVIEWED nullobj
```

Note that at the "bottom" of this construction is the node **nullobj**. This indicates *absence of an overt object.* It is used to fill the object slot on intransitive verbs and also empty object slot in the passive object **passobj**, as in the tree above.

Strings, LXRs and Disjunctive Rules

In general, there are three basic types of rules in string grammar: lxr constructions, string constructions, and disjunctive rules. A disjunctive rule consists of a series of single-element

choices. For example, the object rule is a disjunctive rule, naming all the possible object options, separated by semi-colons (indicating disjunction). The ordering of disjunctions in a rule will affect which parse is found first, since options are applied in order. If the grammar is used with the assumption that the first parse will be the one used, then ordering of options can become important. However, if the system is allowed to run to all parses, then each option will eventually be tried.

```
object ::=
      nstgo;       % noun string object
      vingo;       % present participle + object
      vo;    % infinitive + object
      venpass;% passive object
      tovo; % to + verb + object
      ...
```

The expansion of the **nvar** rule is also a disjunctive rule:

```
lnr ::= ln, nvar, rn.
nvar ::= *n;              % noun
       namestg;    % proper name construction
       *ving;             % gerund
       nulln.             % empty head, e.g., "the few (nulln) are here"
```

By contrast, any rule that has multiple required elements (indicated by ",") is either a string, or it expands an lxr node into left adjunct + head + right adjunct, or it involves punctuation. In general, rules do not mix options (disjunction) and required elements (conjunction). There are, of course, a number of exceptions to this principle, but it is an important one to follow when writing grammar rules, since it preserves clarity and maintains the necessary separation between string definitions, lxr definitions, and disjunctive rules. For the conjunction meta-rule to work properly, for example, it is important to identify string and lxr type definitions.

Empty Elements

One of the unusual features of string grammar is the proliferation of empty elements. Since adjunct slots are included as part of the basic node definitions, the result is that these are often unfilled (indicated by a null) element. There is also **nullobj**, which indicates an empty verb object for intransitive verbs. In addition to these, there are many other flavors of empty elements which carry important information for construction and regularisation of the surface syntax. There is the nulln filler for **nvar**, as in *these three are missing*. There are several kinds of null elements associated with fragmentary input; there is a special kind of null (nullc) for handling gapping in conjunction, and yet another (nullwh) for handling gapping in wh-constructions. Being to distinguish the kind of empty string found in a given location aids the later regularisation and semantic phases in reconstruction of the missing information.

Meta-Rules: Conjunction and Wh

One of the major departures of PUNDIT's Restriction Grammar from Sager's string grammar is PUNDIT's use of meta-rules to capture certain high-level regularities. The conjunction meta-rule mechanism is installed in the current PUNDIT system. It operates on the set of BNF definitions (without conjunction) and produces a new set of grammar rules which cover most cases of conjoining and gapping under conjunction. The meta-rule expands each node of type string or lxr to include, as one option, a conjunction followed by a recursive call to the rule. Thus the expansion for lnr (simplified) is:

```
(lnr ::= ln, nvar, rn) =>
      (lnr ::=    ln, nvar, rn;
```

```
ln, nvar, conj_wd, lnr).
```

Thus the lnr node can either be expanded as usual, or it can invoke the conjunction option, which has a conjunction word followed by a recursive call to lnr. (In actuality, the rule is written more efficiently, so that the ln+nvar+rn does not have to be rebuilt if there is a conjunction.) Thus BNF definitions can be written without worrying about conjunction, as long as nodes are properly classified as lxr or string nodes. The meta-rule component is then applied to generate automatically the correct rules to support optional conjoining.

Wh-constructions (relative clauses, questions, indirect questions, reduced relatives) are also be handled by meta-rules. Here, the function of the meta-rule is be to introduce parameters into each definition, so that gap information can be passed around, namely the need for a gap, or the fact that a gap has been found. This makes the handling of wh-constructions invisible to the grammar writer, who need only worry about routine constructions. The treatment of wh-constructions combines in a very natural way with the meta rule treatment of conjunction.

Naming Conventions

String grammar has a fairly mnemonic set of naming conventions, once you get used to it. For example, objects are named by their components, e.g., tovo = *TO* + *Verb* + *Object*, or pn = *Preposition* + *Noun*. Somewhat confusing is the stg suffix, as in nstg, astg, dstg. Although stg stands for *string*, in fact NONE of the things named by stg are strings. They are lxr constructions. Once you get past that basic confusion (of unknown historical origin), the names are fairly logical.

Type Lists

Since there are certain generalizations associated with strings and lxr nodes, these are captured by *type lists*, which allow the grammar writer to define types, and then to use the associated type names in writing restrictions. For example, there is a type lxr, a type lx, a typerx, and a type string. The type lxr nodes have an operation on them called core, which goes to the head of the lxr construction; this operation is used in restrictions, which often state constraints between heads of syntactic constructions, e.g., between the head of the subject and the head (tensed verb) of the ltvr node for subject-verb agreement. Both the lx and the rx nodes belong to a broader type, the adjunct type. Adjuncts can typically be empty; the adjunct slot is the string grammar mechanism for allowing optional elements.

2. Coverage of PUNDIT's Grammar

The subsection will summarize the current state of PUNDIT's coverage. As mentioned in the introduction to this section, coverage has very much been driven by the needs of the particular domains we have processed. As a result, it is somewhat uneven, although quite broad.

Noun Phrases

Coverage of noun phrases is generally very good. It includes treatment of complex prenominal modifiers: multiple nouns, adjectives, qn expressions such as *a two-foot deep hole*, and nq expressions, such as *the number 2 pump*. Nominalizations are handled as ordinary noun phrases in the syntax, so they are covered and later converted by semantics to capture the underlying verb semantics. A wide range of post-nominal expressions are also covered, including multiple prepositional phrases, participial expressions (*the book read by the students, the person running the race*), adjective expressions (*the student present for the exam*), appositives and parenthetical expressions (*Florence Joyner, the Olympic athlete*, and *my PC (the one I bought a*

month ago). Relative clause coverage has been greatly expanded with the introduction of the new wh-module and includes both standard relative clauses, and zero-complementizer relatives (*the person I saw*). Pronouns are handled by a separate lpror option for the noun phrase; this is done because pronoun take a highly restricted set of left and right adjuncts, compared to nouns.

Adjective Phrases

Coverage of adjective phrases, in pre-nominal position, predicative position and verb complement position is extensive. In predicative and verb complement positions, adjectives can take complex right modifiers, including prepositional phrases (*certain of a fact*) and a variety of clausal complements (*certain that they came, certain to come*). In the left adjunct slot, adjectives can be modified by adverbs, e.g, *very certain*.

Adverbials

The coverage of adverbials in PUNDIT includes left and right modifiers and a recursive definition (e.g., for *very long*).

Verb and Verb Complements

Our current grammar includes more than forty classes of verb complement (object). Selection of the appropriate complement set is controlled by a pruning mechanism that takes the intersection of the verb's subcategorization constraints (given in the *objlist* for the verb entry in the lexicon) with the set of object options. Classes of complement types include:

direct object,
ditransitive,
objects of auxiliary verbs:
    vo (*I may* read the book);
    vingo (*I am* reading the book);
    veno (*I have* finished the book);
    venpass (*She was given the book*);
objects of *be* and other copulative verbs:
    objbe (*They are here/at home; they remain leaders*),
direct object + prepositional phrase,
particle + various object types,
    (e.g., *close up, close up the store, close the store up*),
clausal objects
    (e.g., *I said that I would come; it seemed to be raining*).
equi-verb objects
    (e.g., *I wanted to go*).
small clauses
    (e.g., *they painted the house red*).

Each of these object options has a regularization rule associated with it that allows correct reconstruction of the underlying semantics, including correct handling of subject/object control issues. This is done by the Intermediate Syntactic Regularisation component and will not be further discussed here; see the PUNDIT Guide to Verb Objects for more complete documentation of PUNDIT's object options. One respect in which PUNDIT's treatment of object differs from string grammar is in a uniform treatment of modals, which simply take the object option vo, namely infinitive verb + object.

Sentential Adjuncts

The grammar covers a variety of sentential adjuncts, including adverbial modifiers (adverbs and prepositional phrases), purpose clauses (*I did it to win*), and a range of subordinate clauses (*until finished; before they came; after running the race*). It now also covers a class of adverbial phrases consisting of a lone noun phrase. In normal English, this includes time expressions, e.g., *I left last week.* Also needed for message texts is a similar location adverbial construction, such as *lesion right lung*, where *right lung* is a locative phrase without a preposition. Both of the require strong selectional or semantic constraints, in order to avoid taking almost any noun phrase in any adjunct slot. Not included yet are right-dislocated relative clauses (*the person came whom I wanted to meet*).

Conjunction

The conjunction meta-rule component generates rules to handle conjunction from the basic BNF definitions. Conjoining is allowed only at lxr and string type nodes, which eliminates some of the spurious ambiguity that can be associated with treatments of conjunction. The current mechanism handles a variety of conjunctions (*and, or, but*), paired constructions (*both...and, neither...nor*) and "comma-conjunction" (use of comma to take the place of an explicit conjunction in a list such as *apples, oranges and pears*). Since the meta-rule generates a recursive definition, an arbitrarily long series of conjunctions can be handled.

```
lnr ::= ln, nvar, rn

        =>

lnr ::= ln, nvar, rn;
        ln, nvar, rn, conj_wd, lnr.
```

In addition, the meta-rule component allows for gapping under conjunction. In particular, it can handle gapped subject, gapped object, and gapped verbs, as follows:

I mixed up the batter and baked the cookies.
I cooked and they ate the cookies.
I baked the cookies and Robin the cake.

At the moment, there are certain constructions that are not handled by the current conjunction mechanism. One problem is that conjunction requires homogeneity -- only like objects can be conjoined, for example. Thus PUNDIT cannot parse the construction *my friends and I* because the first conjunct is lnr and the second is lpror. Also, certain kinds of partially gapped objects are not handled, e.g. *they broke through and demolished the plate glass window*, has a gap in the first conjunct that is embedded in the prepositional phrase object, following the the preposition *through*. Thus the object is partially gapped -- which is not currently handled.

Wh-Constructions

The new meta-rule component for wh-constructions now covers questions, relative clauses and indirect questions (*I don't know what they want*). We plan to extend it shortly to cover headless relative constructions (*Whatever you need is here*) as well. It supports the interaction between conjunction (and its gaps) and the wh-constructions (and their gaps).

Fragments

Because much of our work has been focused on message traffic, PUNDIT supports a comprehensive, elegant treatment of fragmentary and run-on sentences that are characteristic of message text. There are five basic fragment types, including fragments for missing subject (**tvo:** *was repaired*), missing verb (**zero_copula:** *disk bad; disk repaired*), missing subject and verb (**predicate:** *broken since yesterday*), missing object (*engineer repaired*), and noun phrase fragment (**nstg_frag:** *bad drive*). Other recently added center string rules include rules for response fragments, necessary to handle certain kinds of question/answer interchanges, e.g., *Are you going? Yes.*

2.1. Debugging Tools and Advice

There are a few tools that are useful in debugging parses that either fail or are incorrect. First, the grammar may be called on any constituent, not just sentences. For example, to see if something parses as a noun phrase, the parser can be called via **parse(nstg)**, which will prompt for input, and will produce parses of all substrings of the input that can be parsed as a noun phrase. This is often useful in a divide-and-conquer approach to debugging, which each phrase can be checked for its parse.

The grammar can be run in two modes: interpreted and translated. When run interpreted, grammar rules are applied as data structures; they do not constitute Prolog procedures, so the normal Prolog spy mechanism does not work. However, restrictions can be spied on, even in interpreted mode. This is often useful to get a sense of how far the grammar has gotten, e.g, if you spy on w_agree, which follows completion of the verb, you know that the verb has been built. Since the parse tree is passed as a parameter to each restriction, spying on a restriction also gives you the current (pretty-printed) partial parse tree, which can be useful. The interpreted mode has a *grind* mechanism available, which allows the user to specify a list of definitions, or all definitions, to be printed out each time they are applied in generating a parse. This is *not* interactive, but can be instructive if one has the patience to follow each application of a set of rules.

When running in translated mode, all grammar definitions are translated into procedure calls. Thus any definition can be traced via the normal spy mechanism. Again, the parse tree is present as a parameter, so you will also be able to see the tree. This is very convenient for debugging.

In addition to these limited tools, there are general strategies for debugging. Step one is to make sure that you are running with selection turned off and with semantics turned off. Either of these can cause unannounced failures when turned on. (They are turned on and off via the switches mechanism -- see the PUNDIT User Guide for information on switches. Step two is to either simplify the sentence or to try the divide and conquer method, parsing constituents one at a time. The idea is to find the problem area in the sentence and to be able to reproduce the bug on a minimal structure. Once you know approximately in what construction the bug is occurring, you can either try spying selected restrictions (and definitions, if running translated) or try the brute force method of grinding. At this point, however, it should be emphasized that debugging the grammar remains an art, rather than a science.

ANNOTATED ALPHABETIC LISTING OF BNF DEFINITIONS Adapted from Sager, Natural Language Information Processing, pp. 310-321 With additional annotations for PUNDIT usage.
** assembled 9/88 by Lynette Hirschman; updated 9/89

********** Annotations:

# indicates NOT in current PUNDIT system
$ indicates in PUNDIT, but NOT in Sager's book.
! indicates significant difference in PUNDIT from Sager's treatment.
****

adjadj recursive definition of pre-nominal ADJectives
defined as: {d_adjadj},lar1, (adjadj;~);
lqnr, (adjadj;~)

! adjinrn ADJective IN RN (right adjuncts of the noun)
PUNDIT handles by astg option in rn.

adjn ADJective + n (noun phrase); permutation of object option na,
as in "painted red the house which I saw last week"

! adjpreq ADJective Pre (i.e. before) Q (quantifier),
handled as q option in lq (left-quantifier adjunct).

andstg and string, to handle conjunction.
PUNDIT handles conjunction differently, via metarule.

and-orstg and/or string for conjunction
PUNDIT handles conjunction differently, via metarule.

apos Adjective POSition of the ordered left adjuncts of a noun
defined as: adjadj; null.

! appos Appositive (in right noun adjunct slot)
Differs from Sager in support of parens, explicit punctuation.
defined as: [,],nstg, ([,];{w_endmark});
[(],nstg,[)]

asobjbe AS + OBJect of BE, e.g., "they served as messengers", or in
passive object option, as in: "she was considered as a candidate".

$ assert_frag
assertion + fragment -- type of center string
defined as: assertion,internal_punct,{vso_selection};
fragment,internal_punct,{vso_selection}

assertion subject + tense + verb + object, with optional sentence
adjuncts between these elements.
defined as: sa,subject,sa,ltvr,{wagree},sa,object,sa

assertions
null assertion + sentence adjunct, for e.g., "they ran and fast".

asstg as string for comparative;
PUNDIT does not yet have a real treatment of comparative,
but currently handles certain constructions via the
conjunction metarules.

! astg Adjective string (not really a string!), for predicate adjective
or adjectives in complement constructions.
in PUNDIT, only defined as lar, not lqnr.
defined as: lar.

as-well-as-stg
AS WELL AS STrinG, for conjunction.
PUNDIT handles conjunction via meta-rule.

avar Adjective VARiant, containing head of adjective construct.
defined as: lcda,*adj
lcda,*ving
lcda,{d_ven_avar},*ven

$ be_aux BE-AUXiliaries -- possible objects following the verb be:
defined as: vingo;
venpass;
tovo

beingo BEING_ Object (as object of be, e.g., "He is being difficult"
PUNDIT handles as be_aux object type.

bothstg both string, for conjunction.
PUNDIT handles conjunction via meta-rule.

butstg but string
PUNDIT handles conjunction via meta-rule.

! center center string of sentence
differs from Sager in addition of fragment, compound options.
defined as: {dquest2},assertion,{w_endmark},{vso_selection};
{dquest1},question,{w_endmark},{vso_selection};
imperative,{w_endmark},{vso_selection};
fragment,{w_endmark},{vso_selection};
compound,{vso_selection}

commastg comma string, used for conjunction.
PUNDIT handles conjunction via meta-rule.
$ commaopt COMMA OPTion, consisting of comma or null.
defined as: ,; null.

$ compound COMPOUND center, consisting of recursive def. of assertion or
fragment, followed by center, or of a runon sentence.
defined as: assert_frag,center;runon

compar Comparative complement (e.g., It is so old that it is
decaying.) PUNDIT has no treatment of comparative at this time.

cpdnumbr Compound number (e.g., one hundred)

csstg CS (subordinate conjunction) STrinG list of options in sa.
PUNDIT handles subordinate clauses as explicit options.

cl.should Subjunctive form of assertion, using untensed verb
defined as: [that],subject,sa,lvr,sa,object,sa

dashstg dash string, for conjunction
PUNDIT handles conjunction via metarule.

dateprep date preposition (e.g., on, in, until, since, etc.)
PUNDIT handles date as a special form of "namestg"
in nvar; dates themselves are handled via the "shapes"
component.

dayyear Various forms of date
PUNDIT handles numerical dates via the "shapes" component.

dpsn Particle (e.g., up, out) + sn (embedded sentence). (e.g., He found out that we went.)

dp1 Particle (e.g., carry on, find out), occurs as object option
defined as: {d_dpval},*dp

dp2 dp (particle) + n (noun phrase), occurs as object option.
defined as: {d_dpval},*dp,nstgo

dp3 n (noun phrase) + dp (particle), occurs as object option.
defined as: {d_dpval},nstgo,*dp

dp4 of-permutation of dp3, e.g., "the splitting up of the project"

dp1p dp1 (particle) + p (preposition), occurs as passive object option
defined as: dp1,p.

dp1pn dp1 (particle + pn (prepositional phrase)
defined as: dp1,pn

dp2pn dp2 (particle + noun phrase) + pn (prepositional phrase)
defined as: dp2,pn

dp3pn dp3 (noun phrase particle) + pn (prepositional phrase)
defined as: dp3,pn

dp4pn of-permutation of dp3 + pn (prepositional phase)

! dstg aDverb string (which is not really a string, however).
in Sager, defined recursively, but not in PUNDIT for now.
defined as: ldr.

eitherstg either string, for conjunction
PUNDIT handles conjunction via meta-rule.

! embeddedq EMBEDDED Question
handled as snwh option in object.

! endmark Punctuation at end of center string of sentence
PUNDIT lists explicit options of "." , "!" and "?" at sentence and ";", "," and "-" connecting centers internally.

$ eqtovo The EQui form of TOVO, where implicit subject is same as the matrix verb subject, e.g., "I want to go", as opposed to the tovo option, "the pump seems to fail", where overt subject is not really subject of matrix verb.
defined as: [to],vo

especially-stg
Especially string, for conjunction.
PUNDIT handles conjunction via meta-rule

fortovo FOR + subject + TO + Verb (infinitive) + Object (e.g., For John to see her is important).

fortovo-n for + to + Verb + Object (less one noun phrase in Object, e.g., the person for John to see), used for wh-gaps.
PUNDIT could handle wh-gaps via meta-rule, although this construction is not yet handled in current meta-rule treatment.

! fraction Fraction; PUNDIT handles via "shapes" component and via fraction_q definition in qvar.

$ fragment option of center, used to parse fragmentary constructions
defined as: tvo;
serocopula;
nstg_frag;
objbe_frag

howqastg HOW + Quantifier (much, many) or Adjective + [of] + article STrinG (e.g., how much of the cake, how good an argument)
defined as: how,([much];[many]), ([of],*t;null)

! howqstg HOW + Quantifier (much, many) STrinG.
Handled as option of dstg creating a wh-construction.

imperative
Imperative sentence in center string
defined as: sa, vo.

$ internal_punct
internal punctuation, separating elements in assert_frag def.
defined as: ',' ; ';' ; '-'

introducer
Pre-Center connective to preceding sentence (e.g., and, or, nor, for)

Not yet in PUNDIT.

la Left adjunct of Adjective
defined as: null;{d_dla},*d

lar Left adjunct of adjective (optional) + Adjective + Right adjunct of adjective (optional)
defined as: la,*adj,ra

! lar1 lar with limited Right adjuncts, as it occurs to the left of a noun;
in Sager, defined as usual lxr, with ra1 "enough" or null
defined as: la,avar

ldate Left adjunct of date
Dates in PUNDIT handled by shapes component.

ldater Left adjunct of date + Date + right adjunct of date
Dates in PUNDIT handled by shapes component.

lcda Left part of Compound Adjective
defined as: null;{d_lcda},*n

lcdn Left part of Compound Noun
Not in PUNDIT

lcdva Left part of Compound Verbal Adjective (e.g., a hog raising farm)

lcs Left adjunct of cs (subordinate conjunction)

$ ld Left adjunct of aDverb, captures recursion in adverb
defined as: null; {d_two_dstgs}, dstg.

$ ldr Left adjuct of aDverb + aDverb + Right adjunct adverb defined (recursively) as lxr construction.
defined as: ld, *d, rd.

! ln Left adjunct of the Noun
PUNDIT omits nspos position of Sager.
defined as: tpos,qpos,apos,npos,{w_np_agree}

lname Left adjunct of a Name (e.g., Dr. Jones)

! lnamer Left adjunct of a name + Name + Right adjunct of name
defined as: lname, *proper, rname.

lname Left adjunct of NAME
defined as: *title;null.

lnamesr Left adjunct of name + possessive form of name

lnr Left adjuncts of the noun + n (noun) + Right adjuncts

of the noun
defined as: ln,nvar,{w_noun_agree},rn,{w_ving_lnr}

lnsr Left adjuncts of the Noun 'S (possession-case noun) + limited Right adjuncts of the noun
defined as: (ln,*ns,{w_noun_agree});[whose].

lp Left adjunct (e.g., adverb) of Preposition
defined as: {d_lp},dstg;qn;null

lpro Left adjunct (e.g, Adverb) of Pronoun (e.g., only he)
defined as: null;{d_dltpro},dstg

lq Left adjunct of Quantifier
defined as: *adj, {w_adj_pre_q};{d_dlq},*d; null

lqnr Left adjunct of Quantifier + Noun string + Right adjunct
defined as: lq,qnpos,rq

lqr Left adjunct of quantifier + Quantifier + Right adjunct
defined as: lq,*q,{w_scope},rq

lt Left adjunct of t (determiner), e.g., "all" in "all the"
defined as: *d; lqr,{w_pre_tpos};null.

ltr Left adjunct of t + t (determiner) + Right adjunct of t
defined as: lt, *t.

ltvr Left adjunct of tensed verb + Tensed form of Verb + Right adjunct
defined as: lv,*tv,rv

lv Left adjunct of V (verb)
defined as: null; {d_lv},*d

lvenr Left adjunct of verb + VEN (past participle of verb) + Right adjunct of verb
defined as: lv,*ven,rv

lvingr Left adjunct of v + VING (-ing form of verb) + Right adjunct
defined as: lv,*ving,rv

lvr Left adjunct of v + verb (infinitive) + Right adjunct of v
defined as: lv,vvar,rv

lvsa Sentence Adjunct occurring to the Left of ving or ven in the adjunct strings vingo and venpass

lw Left adjunct of w + w (the tense or a modal) (e.g., just can't)
PUNDIT handles tense (w) as regular verb class with vo object.

na Noun phrase + Adjective (as object, eg., paint the house red)
defined as: nstg,sa,lar

namepart Name part (all parts of proper name preceding surname)

! namestg NAME STrinG (as value of nvar)
In Sager, defined as lname + *n + rname, where lname
and rname are "name" specific, like titles, "Jr." etc.
This has been used very differently in PUNDIT, to contain
various special kinds of nouns from "shapes" component
defined as: *date; *part_number; lnamer;nq,{w_nq_number}; *time

nasobjbe Noun phrase + AS + OBJect of BE, e.g., "she interpreted it
as a linguist"
Not yet in PUNDIT

nd Noun phrase + Adverb (as object, e.g., put it here)
Not yet in PUNDIT, but needs to be added!

neitherstg
neither string, used in conjunction.
PUNDIT handles conjunction via meta-rule

nn N (indirect object noun phrase) + Noun phrase
defined as: nstg, nstg.

nnn Nouns occurring as left adjuncts of a head noun (e.g.,
herring gull colony). Used recursive definition
defined as: {dnl},*n; namestg; *n,nnn; namestg,nnn

norstg nor string, for conjunction
PUNDIT handles conjunction via meta-rule

notopt optional not
"not" treated as adverb in PUNDIT

npn Noun phrase + prepositional phrase (as object)
defined as: nstgo,pn

npos Noun POSition of the ordered left adjuncts of the noun
defined as: nnn; null

npsnwh Noun phrase + Preposition + snwh (wh-string as a Sentence
Nominalization)
PUNDIT could handle wh-structures via meta-rule but doesn't yet

npsvingo Noun phrase + Preposition + Subject + ving (-ing form of
verb) + Object
Not yet in PUNDIT

npvingo Noun phrase + Prepositional phrase + ving (-ing form of
verb) + Object
Not yet in PUNDIT

npvingstg Noun phrase + Prepositional phrase + vingstg (either
vingofn or nsvingo)

PUNDIT handles vingstg as normal noun construct, whose head is *ving (see nvar definition)

nq Noun phrase + Quantifier/letter (e.g., the Mark 2 analyzer, the Model B spectrophotometer)
defined as: nqnvar,*q

$ nqnvar in PUNDIT, just a regular noun
defined as *n

nsnwh Noun phrase + snwh (wh-string as Sentence Nominalization)
defined as: nstgo, sa, snwh.

nspos Possessive Noun of type position of the ordered left adjuncts of a noun (e.g., one lost children's bicycle)
Not currently in PUNDIT

! nstg Noun string
Currently PUNDIT does not support nwhstg option for, e.g, "what I like is fish"
defined as: {d_endmark},(lnr,{np_selection};
lpror;
nsvingo).

$ nstg_frag Noun-STrinG FRAGment, e.g, "Bad disk drive."
defined as: sa,lnr,sa,{w_bare_nstg}.

nstgo Noun string as Object, used to mark objective case for pronouns
defined as: nstg.

nstgt Noun string of Time, e.g., "last week"
defined as: nstg.

nsvingo N'S (possessive-case noun or pronoun) + VING (-ing form of a verb) + Object
defined as: {d_nullLnsr},vingo;
{d_lnsr},lnsr,vingo,{w_true_vingo}

nthats Noun p[illegible]e + THAT + assertion (verb object option)
defi[illegible] nstgo,sa,thats

ntobe Noun pi[illegible] + TO + BE + object of be
PUNDIT [illegible]s not require distinction between ntobe and ntovo.

ntovo Noun phrase + to + V (infinitive) + Object (verb object option)
e.g., "I expected them to go"
in PUNDIT, this is distinguished from objtovo. Ntovo is for objects where the noun is NOT also an object of the matrix verb.
defined as: subject,[to],vo.

numbstg Number string
Not in PUNDIT; numbers in PUNDIT handled by "shapes" component.

null Empty String
defined as ~ (symbol for the empty string)

$ null_aux NULL AUXiliary verb for zero-copula fragment,
e.g, "disk replaced" => "disk be replaced"
Used to mark missing auxiliary, for regularization.
defined as: ~

$ null_main NULL MAIN verb for zero-copula fragment,
e.g., "disk bad" => disk be bad.
defined as: ~

nulln NULL Noun, used to mark missing head noun in "the three were here"
defined as: ~

nullobj NULL Object, used for intransitive verbs, as in "it broke".
defined as: ~

nullwh NULL WH, used to mark the wh-gap in questions and relative clauses
defined as: ~

nvar Noun or VARiant -- *options of head for lnr construction.*
Includes nouns, names, gerund as noun, and nulln (the
empty noun in e.g., "the three").
defined as: *n; namestg; *ving; {dn2},nulln, {wn1}.

nvsa Noun + Verb Sentece Adjunct of the type: "we know" in,
e.g., "It is, we know, unusual."
PUNDIT does not yet handle this, but will need to spoken input.

nwhstg Noun position WH-STrinGs (e.g., What he cooks tasts good).
Contrast with wh-complements, i.e., sentence nominalizations
snwh, e.g., What he cooks depends on what's on sale.
PUNDIT does not yet handle these, but they can easily be
to the wh-meta-rule treatment.

obes Object of be + tensed form of BE + Subject of be
Used for permuted sentence constructions, e.g., "Smart are they..."
PUNDIT does not yet handle this type of construction.

objbe Predicate noun phrase or adjective phrase of pn or adverb
defined as: astg; nstg; {d_of}, pn.

$ objbe_frag
OBJect of BE as FRAGment, e.g, "down since 10/12".
defined as: sa,objbe,{w_predicate},{w_endmark}

objbesa OBJBE occurring as Sentence Adjunct
Not in PUNDIT.

object The set of Object strings of verbs in active voice
defined as: (npn;objtovo;pnthats;nthats;pnthatsvo;nn;na;pnn;
dp2;dp3;dp2pn;dp3pn;nsnwh;dpsn;ntovo;

pn;nstgo;astg;objbe;nthats,dp1pn;dp1;
thats;assertion;svo;c1should;sven;sobjbe;
dstg;veno;be_aux;eqtovo;tovo;vo;nullobj,
{wverbobj},{w_preobj_sa})

objectbe OBJBE + verbal objects of be
PUNDIT uses be_aux and objbe as options of OBJECT instead.

$ objtovo OBJect TO + Verb + Object construction.
objtovo is distinct from ntovo in that the object serves
both as object of the matrix verb and subject of the embedded
clause.
defined as: nstg,[to],vo

ornot OR NOT, terminating yes_or_no question,
e.g, "Are you coming or not?"
Not currently handled in PUNDIT.

orstg or string
PUNDIT handles conjunction via meta-rule.

p1 Preposition as passive object (see passobj;
e.g., "They can be relied on".
PUNDIT uses *p instead in passive object.

pa Preposition + Adjective (e.g., at last)
Not in PUNDIT at the moment.

parenstg Parenthesis string
PUNDIT handles as option of appos.

particularly-stg
Particularly string, used in conjunction.
PUNDIT handles conjunction via meta-rule.

passobj Object strings in PASSive
defined as: (nullobj;pn;thats;objbe;c1should;assertion;
astg;dp1pn;dp1;snwh;*p;pnthats;pnthatsvo;dpsn;
eqtovo;tovo;dp1p),{wpassobj2})

pdate Date preposition + Date
PUNDIT handles dates by "shapes" component.

permutation
Permuted forms of the center assertion string
Not in PUNDIT yet.

perunit Per + unit (per hour, per cent)
Not in PUNDIT.

pn Prepositional phrase (Preposition + Noun phrase)
defined as: lp,*p,nstg,{w_pval}

$ pnpn Repeated prepositional phrase
defined as: pn,({d_of},pn;~)

pnn Prepositional phrase + Noun phrase (permuted form of npn) in object
defined as: pn,nstgo.

pnsnwh Prepositional phrase + snwh (wh-string as Sentence nominalization)
Not in PUNDIT yet.

pnthats Prepositional phrase + THATS (that + assertion)
defined as: pn, sa, thats.

pnthatsvo Prepositional phrase + THAT + Subject + Verb + Object
e.g., "I asked of them that they leave"
defined as: pn, sa, c1should.

pnvingstg Prepositional phrase + vingstg (either vingofn or nsvingo)
PUNDIT did not define separate vingstg-related options, captures this as nsvingo or *ving in nvar.
Not in PUNDIT

$ predicate PREDICATE fragment, consisting of participle
e.g, "Replacing disk."
defined as: sa, be_aux, {w_endmark}.

psnwh Preposition + snwh (wh-string as Sentence Nominalization)
PUNDIT does not yet handle snwh constructions, but will once wh component is installed.

pstg A subset of prepositional object strings used in the lexicon
PUNDIT does not group these options together.

psvingo Preposition + SVINGO (Subject + ving (-ing form of verb) + Object)
PUNDIT does not handle this now.

pvingo Preposition + vingo (ving + Object)
PUNDIT will handle as nsvingo or *ving in nvar.

pvingstg Preposition + VINGSTG (either vingofn or nsvingo)
PUNDIT will handle as nsvingo in nvar in pn.

pwhnq Preposition + WH-containing Noun phrase + yes-no Question (e.g., From which side did they enter?)
PUNDIT can handle in new wh meta-rule treatment, but doesns't yet.

pwhnq-pn Preposition + WH-containing Noun phrase + yes-no Question less a PN (prepositional phrase) in Object (e.g., To whom is it attributed?)
PUNDIT could handle in new wh meta-rule treatment.

pwhns Preposition + wh-containing Noun phrase + assertion (e.g., "the girl from whose apartment it was taken")

PUNDIT could handle in new wh meta-rule treatment.

pwhns-pn Preposition + WH-containing Noun phrase + assertion less a PN in object (e.g., the artist to whom it is attributed)
PUNDIT could handle in new wh meta-rule treatment.

pwhq Preposition + WH-word yes-no Question
(e.g., "For whom was it ordered?")
PUNDIT could handle in new wh meta-rule treatment.

pwhq-pn Preposition + WH-word + yes-no Question less a PN in Object (e.g., On what is it based?)
PUNDIT could handle in new wh meta-rule treatment.

pwhs Preposition + wh-word + assertion

pwhs-pn Preposition + WH-word + assertion less a PN in object
PUNDIT could handle in new wh meta-rule treatment.

q-assert Assertion used in analysing comparative
PUNDIT does not currently handle comparative.

q-conj Body of conjunction string following a coordinate.
PUNDIT handles via conjunction meta-rule.

q-invert INVERTed assertion used in analysing comparative
PUNDIT does not currently handle comparatives.

qn Quantifier + Noun (where Noun = name of a unit: " a 3-inch line"
defined as: lqr, {d_sing},*n.

qnrep Repeated qn sequence (4 lb. 2 oz.)
Not in PUNDIT yet.

qns Quantifier + possessive NOUN ("a 4 month's history of headaches")
Not in PUNDIT yet.

q-of q-word (e.g., tens, dozens. lots, hundreds + of)
Used in parsing numbers.
Not in PUNDIT.

! qnpos Position of the qn string and nq string in the ordered left adjuncts of a noun, e.g., "a two ton brick";
in PUNDIT, only qn allowed.
defined as: qn

q-phrase ever, usual, necessary in comparative
(e.g., "We will wait as long as usual.")
PUNDII does not handle comparatives.

qpos Quantifier Position of the ordered left adjuncts of a noun
defined as: lqr;null.

question Question as center string of a sentence
defined as: yesnoq; wh_question.

qvar Quantifier Variant in lqr definition, including q, and numbers
numbers handled by shapes in general.
defined as: *q; fraction_q.

ra Right adjuncts of an Adjective
defined as: null;
pnpn;
{d_raising_adj},tovo;
{d_equi_adj},tovo;
d_sent(thats;assertion).

ra1 enough or null as Right adjunct of an Adjective
occurring as left adjunct of a noun
Not in PUNDIT; lar1 has no right adjunct.

rd Right adjunct of an Adverb, e.g., "enough"
defined as: null.

rdate Right adjunct of Date
PUNDIT handles dates via "shapes" component.

$ rel_clause
Takes the place of Sager's rnwh options.
defined as: whRC,assertion,{w_need_gap}

! rname Right adjunct of a Name, e.g., "Jr.", "III"
defined as: null (for now).

rnp Strings beginning with a Preposition as Right adjuncts
of a Noun phrase
PUNDIT does not use this intermediate node, has pnpn option instead
! rn*r Right adjuncts of a Noun phrase (*r indicates adjunction
is repeatable)
PUNDIT does not support repetition except via pnpn rule.
defined as: {d_endmark},pnpn;vingo;
{dn_comp}, (thats;c1should;tovo: sub1;
appos;astg,{w_heavy_rn};rel_clause;
sero_comp;
null;
venpass,{w_heavy_rn}
for wh, also defined as:
{d_endmark},pnpn;

rnsubj Right adjuncts of a Noun SUBJect at a distance in sa
(e.g., A procedure is described which...)
Not yet in PUNDIT.

rnwh Relative clause, i.e., WH-string, as Right adjunct of a Noun

This functionality is captured via options in rn.

$ rpro Right adjunct of PROnoun
PUNDIT distinguished lpror from lnr.
defined as: null.

! rq Right adjunct of a quantifier, e.g., "enough" or empty.
defined as: null

rsubj Roving adjuncts of the Subject (or a more proximate noun) of quantifier type (e.g., We are all amazed).
Not in PUNDIT.

$ runon RUNON sentence or sentence fragments
defined as: assertion,center,{vso_selection};
fragment,center

! rv*r Right adjuncts of a Verb (*r indicates adjunction is repeatable)
Not repeatable in PUNDIT; Sager also allows dstg, pn, qn, sn.
defined as: null.

rw Right adjunct of w (the tense or a modal) (e.g., He is not coming; she will not be here)
PUNDIT does not distinguish modals from regular verbs.

saconj Sentence Adjunct following a coordinate CONJunction
Handled as simple sa in PUNDIT.

! sa*r Sentence Adjuncts (*r indicates adjunction is repeatable)
Not repeatable in PUNDIT; also fewer options.
Specifically, the options for time nstg (nstgt), roving adjuncts (rsubj, rnsubj), passive (e.g, "attacked by the snakes") and comparatives are missing in PUNDIT.
defined as: null;
{d_endmark},commaopt,{dsa},
(({d_d_or_p},dstg);tovo;sub7;sub1;sub0;{d_of},pn;
({d_init_sa},vingo)),
{wmed_sa},commaopt,{w_comma_symmetry}
for wh, also defined as:
{d_post_obj}, {d_nullwh_in_sa},dstg.

sasobjbe Subject + AS + OBJect of BE
Option of object, e.g., "they saw this as their opportunity"
Not in PUNDIT yet.

sawh WH-strings in the set of Sentence Adjuncts
Will eventually be handled via meta-rule.

sawhichstg
WHICH-STrinG (relative clause) as Sentence Adjunct (e.g., "She left, which surprised him.")

sas Adjunct of a Zeroed sentence under conjunction

(e.g., He left, and fast.)
Not in PUNDIT.

scalestg Scale string in qn, e.g., "two feet long"
Not in PUNDIT

! sentence Introducer + center + endmark in Sager
No introducer in PUNDIT.
defined as: center, ([.];[?]).

sn Sentence Nominalisation option of subject,
including thats, fortovo, tovo clshould and snwh
Not included in PUNDIT yet.

s-n Assertion less one Noun phrase (i.e., headless relative clause)
Will be handled by meta-rule in PUNDIT when wh is installed.

! snwh WH-string as a Sentence Nominalisation (i.e., wh-complement)
e.g., "whether I will leave is unclear"
defined as: (whQ,assertion);(whQ,tovo).

sobjbe Subject + Object of BE option of object
e.g., "they consider them fools".
defined as: nstg,sa,objbe,sa.

sobjbesa Subject + OBJect of BE occurring as Sentence Adjunct
Not in PUNDIT.

stovo-n Subject + TOVO-N string as object of have
e.g., "I have things to do"
Not yet handled in PUNDIT

subject Subject of verb in the same string
defined as: nstg; there_def.

! sub0 Subordinate conjunction + Object of be
e.g., "after failing the test"
defined somewhat more broadly in PUNDIT, including Sager's
sub0, sub2, sub3, sub4 definitions.
defined as: *cs0,venpass;
*cs0,vingo;
*cs0,objbe.

sub1 Subordinate conjunction + assertion
e.g., "because they are leaving"
defined as: *cs1,assertion.

sub2 Subordinate conjunction or as or than + venpass (passive
verb with its passive object)
PUNDIT captures in sub0

sub3 Subordinate conjunction + ving -ing form of verb) + Object
PUNDIT captures in sub0

sub4 Subordinate conjunction + ving string (either vingofn or nsvingo)
ving string handled in nstg in PUNDIT.

sub5 Subordinate conjunction + svingo
e.g., "despite the disk failing the test"
Not included in PUNDIT.

sub6 Subordinate conjunction + sobjbe
e.g., "with them out sick"
Not handled in PUNDIT

sub7 Subordinate conjunction + sven
e.g.,"with the crisis ended"
defined as: *cs5,sven.

sub8 Subordinate conjunction (as) + inverted Assertion
Not in PUNDIT.

sub9 Should + svo, subjunctive adjunct
e.g., "should she accept, she can start tomorrow."
Not in PUNDIT.

sven Subject + passive verb with its passive object (venpass)
option of passive object, e.g., "I got the disk fixed"
defined as: subject,sa,venpass,sa.

svingo Subject + VING (-ing form of verb) + Object
Option of object, e.g, "I watched them running the race"
Not yet in PUNDIT.

svo Subject + Verb (tenseless) + Object
Option of object, e.g., "I let them go"
defined as: subject,sa,lvr,sa,object,sa

tense Position for tense-word (modal)
PUNDIT handles modals as regular verbs.

thanstg THAN STrinG, for comparative constructions.
PUNDIT has no treatment of comparatives

thats THAT + assertion option of object
e.g., "I hope that they come"
defined as: [that],assertion.

! thats-n THAT + assertion less one Noun phrase (relative clause with word that instead of wh-word)
PUNDIT handles via meta-rule component for wh.

$ there_def There (pleonastic) option in subject.
defined as: [there].

! title A Title used as part of a name (e.g., Mr., Ms.) in namestg
defined as as atom, option of lname.

tobe TO + BE as tenseless Verb + Object
PUNDIT covers as part of tovo options.

tostg TO string (from 3 to 4 hours)
PUNDIT handles conjunction use of "to" via meta-rule.

! tovo TO + tenseless Verb + Object
Option of verb object, e.g, "She seemed to win".
tovo in PUNDIT is split into tovo and eqtovo, to distinguish
equi cases ("I hope to win") from the raising case.
defined as: [to],vo.

tovo-n TO + tenseless Verb + Object less one Noun phrase in object
e.g., "the person to see"
PUNDIT should handle this via a meta-rule consistent with
meta-rule wh treatment, but does not at this time.

! tpos t (The) Position of left adjuncts of noun phrase
Sager uses ltr instead of just *t..
defined as: ltr; null.

tsubjvo Tense + Subject + tenseless Verb + Object,
e.g.,"Would they were gone".
Not defined in PUNDIT.

$ tvo Tensed Verb + Object fragment
e.g, "fixed the disk"
defined as: sa, ltvr, sa, object, sa.

veno VEN (past participle of a verb) + Object option of object,
e.g., "They had seen the light".
defined as: {dsel4},lvenr,sa,object,sa.

venpass VEN (past participle of a verb) + Passive object
Option of object, e.g, "it was given to her"
defined as: {dsel4},lvenr,{wpassobj1},sa,passobj,sa

verb tensed or tenseless Verb with optional left and right adjuncts
Replaced by ltvr or lvr in PUNDIT.

verb1 tense-word or tensed be or have in question
replaced by ltvr in PUNDIT.

verb2 2nd Verb position in Question
replaced by lvr in object in PUNDIT.

vingo VING (-ing form of Verb) + Object
Option of object, e.g., "I am going to work"
defined as: {dsel5},lvingr,sa,object,sa

vingofn VING + of + Noun phrase
PUNDIT handles as regular nstg.

vingstg VING string (nsvingo or vingofn)
PUNDIT has separate nsvingo definition in nstg.

vingstgpn VING string + pn (presositional phrase)
PUNDIT has separate nsvingo definition in nstg.

vo tenseless Verb + Object
option of object, e.g., "I would do it".
defined as: lvr,sa,object,sa

! vvar Verb VARiant
In Sager, defined as tensed or tenseless verb;
In PUNDIT, used for empty verbs in fragment definitions.
defined as: *v;{d_nullv},null_main;null_aux.

$ wh WH word; note that this calls nstg, which in turn calls wh_word.
defined as: where; when; why; {d_wh}, nstg.

$ whRC WH word for relative clauses
defined as: wh_word; that.

$ whQ WH word for questions
defined as: {d_wh_how},dstg; [what]; wh.

$ wh_word WH word within noun phrases
defined as: [who]; [whom]; [which].

whats-n WHAT + assertion less one Noun phrase
In PUNDIT,could be handled as part of meta-rule treatment for wh.

whens WHEN or where or null + assertion
(when can be null if string adjoins time noun)
PUNDIT handles via meta-rule wh treatment.

wheres WHERE + assertion
PUNDIT handles via meta-rule wh treatment.

wheths WHETHer or where or when or how or why of if + assertion
+ optional [or not]
Not yet incorporated, could be part of wh meta-rule treatment.

whethtovo WHETHer (or other wh-words) + TO + Verb + Object
e.g., "whether to go or not"
Not yet incorporated in PUNDIT, could be handled via wh meta-rules

whevers-n WH-EVER (whose, whenever, whichever, whatever) + assertion
missing a Noun phrase, e.g. " whatever they wish"
Not yet incorporated in PUNDIT, could be handled via wh meta-rules

whln wh-word (whose, which, what, how string) as Left adjunct of a Noun
option of tpos.
defined as: which; what; howqastg.

whn Noun phrase or vingofn string carrying a WH-word
(e.g., whose book was lost)
Handled via whln construction and wh-treatment

whnq-n WH-containing Noun phrase + yes-no question less Noun
(e.g., whose book have you?)
Handled via whln construction and wh-treatment

whns-n WH-containing Noun phrase + assertion less one Noun phrase
Handled via meta-rule in PUNDIT.

! whq WH-word + yes-no question
Handled via meta-rule in PUNDIT.

! whq-n WH-word + yes-no question or assertion less one Noun phrase
Handled via meta-rule in PUNDIT.

! whs-n WH-word + assertion less one Noun phrase
Handled via meta-rule in PUNDIT.

yesnoq yes-no question (e.g., Have you a book?, Did she leave?)
defined as: sa,ltvr,sa,subject,{w_sai},sa,object,sa

$ zerocopula
fragment with ZERO COPULA, e.g, "disk bad".
defined as: sa,subject,sa,lvr,{w_frag_verb},
sa,object,{w_pn},{w_nonnull_ln},sa

$ zero_comp Zero-complement relative clause construction,
option of rn, as in "the person I saw"
defined as: subject, {w_zero_comp},sa, ltvr,{wagree},sa,object,sa.

Lexical Look-Up Procedure
in PUNDIT

Lynette Hirschman

This document describes the lexical look-up procedure for PUNDIT. We begin with a brief description of the lexicon and its organization. We then provide an overview of the functions of the lexical look-up procedure. Finally, we describe in more detail the specific relations used to implement the lexical look-up procedure. Appendix 1 provides a detailed description of the format of a lexical entry.

1.1. Organisation of the Lexicon

The PUNDIT lexicon has several features that are relevant to this discussion.

Entries indexed on first word

Each lexical entry is entered into the (Prolog recorded) database, indexed on the *first word*. Most entries, of course, have only one word; however, for multi-word expressions (e.g., *red blood cell*), the entry is indexed only on the first word (*red* in this example).

Form of entry in lexicon

The colon (both in prefix and infix forms) is used as a functor in the lexicon. Each entry in the lexicon consists of the WORD, the index term, the root, and the attribute list. The source form of the lexicon looks as follows:

:(WORD, root: ROOT, ATTRIBUTE_LIST).
where ATTRIBUTE_LIST is a list of the form:
[LEXICAL_CLASS : ATTRIBUTES | MORE_ATTRIBUTES].

Idioms (multi-word expressions) are entered by use of the circumflex infix operator (^), which connects the words in the multi-word expression, e.g.,

:(red^blood^cells, root: red^blood^cell, [n: [ncount1, plural]]).

The colon is treated as a regular Prolog relation; code associated with its definition causes the source entry to be recorded in the database, indexed on the word (or first word, in a multi-word expression), e.g.,

records(red, :(red^blood^cells, root: red^blood^cell, [n: [ncount1, plural]]).

For purposes of editing and displaying lexical terms, each word is also cross-indexed under its root. This is done by code in the module readin.pl.

Compression of redundant information

The PUNDIT lexicon enters each morphological variant as a separate entry, since there is (currently) no separate morphological component. As a result, there is a great deal of redundancy between morphologically related entries. To minimize this redundancy, the lexicon compresses information, storing the full set of attributes in the *root* entry, and using pointers to this information in the morphological variants. This means that at lexical look-up time, the look-up procedure must "reconstitute" entries for individual words into their full form. This process is described in some detail in section ??. For example, the entry for the word "cells" is as follows:

:(cells, root: cell, [n: [plural, 11]]).

In this entry, *11* is the pointer to the attributes associated with the noun entry. (The use of numbers as pointers is an historical artifact, based on the representation used in the Linguistic String Project; it could and probably should be replaced with more mnemonic pointer labels, such as *noun_attributes, verb_attributes*, etc.). In order to track down the information represented by the pointer *11*, the look-up procedure goes to the entry corresponding to the root (e.g., *cell*) and finds there a specification of what the pointer *11* stands for. By convention, the pointer definition follows (occurs to the right of) its

invocation in a definition (for the root word). For a non-root word (a word which differs from its root), the definition of the pointer may either be found locally, or can be found associated with the root entry. Thus "cell" is a root word, and the definition for "11" is found following its invocation:

:(cell, root: cell, [n: [singular, 11], 11: [ncount1]]).

Using this information, the entry for *cells* is reconstituted as:

cells : [n :[root: [cell], plural, ncount1]].

This is the form returned by **assembledefns**, for ease of use in attaching terminals to the parse tree. When a word is actually attached, only the particular definition corresponding to that terminal is attached to the tree.

Multiple Entries

A single word may have multiple entries in the lexicon. This can reflect incremental additions to the lexicon, or it can reflect differing forms, e.g., different parts of speech, as in the noun *train* vs. the verb *train*; it can result from genuine homographs, such as the verb *can* used as a modal (*be able*) or as a transitive verb for the canning process. At times, it can also reflect an error, where two people have independently entered the same word into the lexicon. In any case, one function of the lexical look-up procedure is to amalgamate these entries into a single entry for purposes of parsing. Where two entries are identical, the program is smart enought to simply collapse them. In other cases, the union of the attributes is recorded. For example, suppose the entry *slow* has the following two entries, one for the adjective and the one for the verb:

:(slow, root: slow, [adj]).}
:(slow, root: slow, [tv: [...], v:[...]]).

During lexical look-up, these are amalgamated into a single entry:

```
:(slow :[
        adj: [root: [slow]],
        tv: [root: [slow], plural, objlist: [...], ...],
        v: [root: [slow], objlist: [...], ...]]).
```

If a word has two *identical* definitions, the redundant information is suppressed. However, if two not-quite-identical definitions are given, they will both be passed along. For example, if the source lexicon contains the following two entries:

:(sugar, root: sugar, n: [singular, mass]).
:(sugar, root: sugar, n: [singular, ncount1]).

then the lexical look-up procedure will generate the following entry for consumption by the parser:

```
sugar: [n: [root: [sugar], singular, mass],
        n: [root: [sugar], singular, ncount1]].
```

Shapes: a grammar for productive forms

The last issue concerns the problem of how to store productive forms in the lexicon. This arises, for example, for numbers, dates, times, part numbers, etc. The solution in PUNDIT is to use a shapes grammar (in shapes.pl), which parses the tokens within a productive form, identifies the class (and attributes) of the lexical entry from the shape of its tokens, and assigns it a definition on this basis. Definitions derived from the shapes component are then added to the list of possible definitions for a word.

Choosing a definition

At this pont,, definitions sharing the same root have been merged into a single definition; however, there may be distinct entries due to distinct roots, or due to idiom look-up, or due to use of the shapes component. The final stage is to chose one of these definitions to

pursue, and hand off the remainder of the word stream for further processing. (In a bottom-up system, it would be possible to generate a lexical lattice at this point, with arcs spanning one or more nodes, and each arc associated with a distinct definition.) For now, the choice of definition done by "longest first". This means that if, for example, there are entries for both "sickle cell" and "sickle cell anemia", if the word stream matches "sickle cell anemia", this definition will be chosen in preference to the shorter sequence "sickle cell". However, this choice is backtrackable, so that if no parse is obtained, the system can backtrack to this point and try a shorter (or different) expression. In general, however, it appears to be the case that if a parse is obtained with the longer definition, it is incorrect (and can lead to spurious ambiguities) to backtrack and obtain multiple parses. Therefore, it would probably be appropriate to introduce some code to commit to this choice in case a parse is obtained.

1.2. The Code in Lexical Look-up

This section documents the important procedures used in lexical look-up. The comments reflect the current state of the code, which clearly could use some clean up.

assembledefns(+InputWordStream,-DefinitionList,-RemainingWords)

This is the top-level routine, called after the call to **makeWordList** has converted tokens into words (code in **reader.pl**). It is called recursively, consuming one lexical unit on each call. A lexical unit is a single word or a multi-word expression that starts at the current point and spans one or more "words". The procedure **assembledefns** has the following steps:

1. Find in the lexicon ALL ENTRIES beginning with WORD
   (done in lookup/2)
2. Match multi-word expressions beginning with WORD
   (done in possible_entries/3);
   this creates a list of possible sequences matching the input stream,
   together with a notation of how many words each candidate eats up.
   creates a data structure e(Def,Num), where Num is number of words - 1
   spanned by the definition.
3. Get all the roots associated with each candidate
   (done in allow_mult_roots/3);
   this changes the "e(Def,Num)" data structure to "e(Def,Root,Num)".
4. Use the roots to decompress the definition (the "number lists")
   (done in fill_in_def/2);
   this creates a set of decompressed possible definitions;
   it also changes the "e" data structure from
   e(Def,Root,Num) to e(RevisedDef, Num),
   where Def = :(Word,root:Root,Attributes)
   and RevisedDef = :(Word, LexClassList),
   where each element of LexClassList =
   LexClass: [root:[Root], LexClassAtts].
5. Merge entries for a given word and same root into a single entry
   (done in merge_entries/2);
   this allows, for example, creation of a single def. given
   two entries, one for slow: [adj], and one for slow: [v, tv];
6. See if WORD is parsable as a shape
   (done in all_shape_entries/2, modify_shape_entries/2);
   this produces an additional list of definitions,
   which is appended to existing list;
7. Select the LONGEST definition

(done by choose_def/4) ** this is a backtrack point **
8. Call assembledefns/3 recursively to process rest of word stream.

lookup(+Word, -ListOfDefsStartingWithWord)

The procedure *lookup* consults the lexicon for all entries stored under **Word** and returns all distinct definitions found under the key **Word** that start with word. This may include multi-word definitions and multiple definitions with either the same or different root forms.

possible_entries(+ListOfDefsStartingWithWd,+RemainingWds,-ListOfMatchingDefs)

This procedure takes the list of possible definitions generated by lookup and tries to match multi-word expressions against the input stream. It generates a data structure e(Def,Num), where *Num* is the number of additional words consumed from the input stream. This is eventually used to the longest multi-word expression from competing possible alternatives.

allow_mult_roots(+ListOfMatchingDefs,+TempList,-RevisedEStructureList)

This procedure takes the output of possible_entries, namely *ListOfMatchingDefs*, and generates extra entries for any definition that is not its own root, but points back to a root definition that has multiple entries. It creates a list element for each entry paired with a specific root definition. It also revised the *e* data structure to have the form *e(Def,Root,Num)*.

fill_in_def(+RevisedEStructureList, -FilledInDefList)

This procedure handles the "decompression" of pointers into explicit attribute lists. Its input is the revised "e" structure list from allow_mult_roots. Its output is a differently structured definition list, with pointers replaced by attribute lists. The output list is structured for ease of use in parsing. Thus the list consists of the word or words, followed by the list of lexical classes. Within each lexical class, we find the root and the remaining attributes associated with that lexical class. Thus the definition list now has the form:

```
Word: [Lex_class1: [root: [Root1] | Lex_class_att_list1],
       Lex_class2: [root: [Root2] | Lex_class_att_list2],
      ...
      ].
```

The procedure fill_in_def calls on fill_in_attrbs, which works right to left and has responsibility for both capturing pointer definitions (and collecting them for use in resolving pointer references) and resolving invocations of pointer definitions, either by looking at those pointer definitions already captured, or by finding the root, and capturing the definitions from the root word.

merge_entries(+FilledInDefList,-MergedDefList)

The procedure merge_entries merges all definitions consuming the same number of words into a single entry. In addition, it merges entries with identical roots into a single lexical-class entry. For example, it would convert the following input to a single entry, first by combining the two entries for word1^word2, then by combining the attribute lists for the entries with identical roots.

[e(word1^word2: [n: [root: [word1^word2], ncount1]], 1),

```
    e(word1^word2: [n: [root: [word1^word2], mass]], 1)]
==>
    [e(word1^word2: [n: [root: [word1^word2], ncount1, mass]]].
```

all_shape_entries(+WordStream, -ShapeDefList)

This procedure invokes the **shapes** grammar against the input stream and produces a set of possible pairs consisting of a shape length and its definition. The shapes grammar is defined in **shapes.pl** and provides entries for productive forms, such as numbers, dates, etc. If the shapes grammar produces no entries, the empty list is returned.

modify_shape_entries(+WordNumShapeEntryPairs, -ModifiedShapes)

This procedure takes as input a list of pairs of the form *ShapeLength-ShapeDef* and returns the appropriate "e" structure list, so that the shapes definitions can be merged with the previously collected definitions.

choose_def(+ListOfPossibilities, -ChosenDef, +WdsAfterStartWd, -RemainingWds)

The procedure **choose_def** takes as its input the merged set of definitions from the regular lexical look-up procedure and from shapes and selects the definition spanning the longer number of words. It also creates a back-track point, so that the remaining definitions can be explored via backtracking into lexical look-up if desired.

APPENDIX 1

SPECIFICATION OF LEXICAL ENTRIES IN PUNDIT

François Lang

This is an attempt to formalize Pundit's lexical entries, which I have coded as part of an error-checking mechanism to be added to the `readin.pl` file. The reason for doing this is that there is currently no mechanism for ensuring the well-formedness of lexical entries which are read in. In fact, as I'll point out, there are a disturbingly large number of lexical entries currently in some lexicon file which, for one reason or another, are bogus.

All terminology set in *slanted font* is defined in what follows.

The following must be true of a Pundit lexical entry:

1. It is a (syntactically correct) ground Prolog term.
2. Its principal functor/arity is `:/3`.
3. Its first argument is a *lexical item*.
4. Its second argument is a term of the form `root:Root`, where `Root` is a *lexical item*.
5. Its third argument is a *definition list*.

A *lexical item* is either a *lexical atom*[1] or an *idiom*.

A *lexical atom* is one of the following:

1. an atom containing exactly one character `C` such that if `A` is ASCII equivalent of `C`, the goal `singleCharacterWord(A)` succeeds.[2]
2. an atom containing only the following characters:
   (a) alphanumerics (i.e., `a` ... `z`, `A` ... `Z`, and `1` ... `9`),
   (b) the single- and double quote characters ("'" and """), and
   (c) the underscore character ("_").

[1]Throughout this specification, I have tried to be very careful to distinguish *atoms* and *atomic terms*. The distinction is that numbers are atomic terms, but not atoms. I.e., if `X` is currently instantiated to a number, the goal `atomic(X)` succeeds, but the goal `atom(X)` does not. I specify here that lexical items are atoms, and not atomic terms, because numbers are now analyzed by the shapes component, and have been taken out of the lexicon.

[2]The predicate `singleCharacterword/1` is defined in the file `reader.pl`.

An *idiom* is a term of the form `X^Y` where `X` is either an integer or a lexical atom, and `Y` is either an integer, a lexical atom, or itself an idiom. E.g., `starting^air^compressor` is an idiom, as is `:(cgn^(-)^25,root:bainbridge,[proper])`.

A *definition list* is a (possibly empty) list of *definition terms*.

A *definition term* is one of the following:

1. A *category definition*,
2. A *pointer definition*,
3. A *lexical category*.

A *category definition* is a term of the form `Cat:FeatureList`, where

1. `Cat` is a *lexical category*, and
2. `FeatureList` is a *feature list*.

A *pointer definition* is a term of the form `Pointer:Definition`, where

1. `Pointer` is an integer, and
2. `Definition` is a *feature list*.

A *lexical category* is a term `C` such that the goal `get_type(C, atomic_node)` succeeds (e.g., `adj`, `n`, `p`, `pro`, `proper`, `q`, `v`, `ven`, `ving`). All lexical categories are atoms.

The `Cat` and `Pointer` terms appearing as the left-hand arguments of `:/2` in category definitions and pointer definitions, respectively, can be referred to as *definition heads*.

There are a few additional constraints:

- At least one definition term in a non-empty definition list must be either a category definition or a lexical category. In other words, it is incorrect for all definition terms in a definition list to be pointer definitions. E.g., the following list is not a valid definition list:

  `[12:[vendadj,h-modal],3:[sasobjbe,nstgo,vingo],11:[ncount1,nonhuman]]`

- A pointer definition in a definition list must appear after (i.e., to the right of) all references to it. This means that pointer definitions should in general appear after all other definition terms in a definition list. E.g., neither of the following lists is a valid definition list:

  `[v:[12],12:[objlist:[nstgo]],tv:[12,plural]]`

  `[v:[13],tv:[13,plural],15:[pval:[for,to]],13:[objlist:[nstgo,npn:[15]]]]`

A *feature list* is a (possibly empty) list of *feature terms*.

A *feature term* is one of the following:

1. A term of the form `Feature:Expansion`, where
   (a) `Feature` is a *feature head*, and
   (b) `Expansion` is a *feature list*.
2. A *feature*.

A *feature head* is one of the following:

1. A *lexical attribute*
2. A term of the form `X-Y` (leftover LSP medical categories), where `X` and `Y` are both atoms. In such terms, `X` will almost always be the atom `h`. A complete listing of all such "hospital terms" currently appearing in Pundit lexicons has been collected. Note that these `X-Y` terms are not atoms, contrary to popular belief and expectations. These `X-Y` terms are totally irrelevant to all current uses of Pundit, and could (should) be removed from our lexicons. Removing them would simplify this formalization of lexical entries.

A *feature* is one of the following:

1. A term of the form `X-Y` as above
2. An *idiom*
3. An atomic term[3]

A *lexical attribute* is any one of a well-defined set of atoms (such as `objlist`, `pobjlist`, `pval`, and `dpval`).

As mentioned earlier, there are currently a number of lexical entries which do not meet these criteria. Some of them have problems not directly related to their form. For example, many lexical entries assume that `cs2`, `cs3`, `cs4`, `cs5`, `cs6`, `cs7`, `cs8`, `int`, `punct`, and `w` are lexical categories. However, since none of these atoms appears prefixed by '`*`' in the body of a BNF grammar rule, they are not known as atomic nodes, and thus not lexical categories. If such

[3]We cannot restrict features which are not idioms and not "hospital terms" to be just lexical attributes, since both numerical pointers and lexical items (prepositions, for example), neither of which which are lexical attributes, regularly appear as features in feature lists, as in `n:[singular,11]` and `pn:[pval:[off,from]]`. It might be possible somehow to restrict non-idiom features to integers, lexical items, and lexical attributes, but this will require more thought. For now, we will say no more than these features are atomic.

an unknown lexical category is encountered while reading in a lexicon, an appropriate and persipcuous warning message should be issued, but reading in the lexicon should be allowed to continue.

In addition, there are the following entries (at least) which are simply bogus. Again, if such an ill-formed entry is encountered while reading in a lexicon, a warning message should be issued, but processing should continue. For most of these entries, it is left as an exercise to the reader to determine the exact problem!

```
:(re-examination,
  root:examination,
  [n:[11,singular],11:[nonhuman,h-vmd,h-rep,h-record]])

:(re-examinations,
  root:examination,
  [n:[11,plural]])

:(seeking,
  root:seek,
  [ving:[12],vveryving])
```

Hint: vveryving is not a lexical category.

```
:(shear,
  root:shear,
  [v:[12],tv:[12,plural],
   12:[objlist:[nstgo,nullobj,[dp],npn:[pval:[off,from]]]]])

:(sheared,
  root:shear,
  [tv:[12,past],ven:[14],
   14:[12,pobjlist:[nullobj,[dp],pn:[pval:[off,from]]]]])

:(timing,
  root:time,
  [n,singular,ving:[12]])

:(try,
  root:try,
  [n:[11,singular],v:[12],tv:[12,plural],
   12:[objlist:[3],notnobj:[1],vendadj,h-modal],
   3:[sasobjbe,dp4:[15],dp2:[15],dp3:[15],nstgo,vingo,eqtovo,npn:[16],nullobj],
   16:[pval:[on]],1:[ntime1,ntime2],14:[objlist:[3],vendadj,pobjlist:[4]],
   4:[asobjbe,dp1:[15],pn:[16],nullobj],15:[dpval:[out]],
   11:[ncount1,nonhuman,h-_1882]])
```

Hint: Look at the very end of the entry. Also, the pointer definitions for 3 and 16 appear before they are referenced.

```
:(works,
  root:work,
  [n,singular,tv:[12]])


:(nimitz,root,nimitz,[proper])
```

Prolog Structure Editor

Leslie Riley
John Dowding

LBS Technical Memo 29
January 1986

The Prolog Structure Editor is a general structure editor written in Prolog. It is intended to make it easy to edit Prolog terms by allowing the user to edit a term by traversing its internal structure. As used in the Natural Language group, the Prolog Structure Editor allows you to edit grammar rules, word definitions in the lexicon, and arbitrary Prolog clauses. You may invoke the editor on one of these three types of structures by using one of the following top-level procedures:

edit_rule(Key)
edit_word(Word)
edit_clause(Functor)

The *edit_rule* procedure takes as its argument the name of a non-terminal. It then allows you to edit all of the grammar rules that define that non-terminal. In order to maintain consistency between the grammar rules and their translated versions, when you have completed editing the set of rules the editor will ask if you want to re-translate them.

The *edit_word* procedure takes as its argument a word from the lexicon. It then allows you to edit the definitions of that word and all of its morphological variants.

The *edit_clause* procedure takes as its argument the name of some Prolog procedure. The editor finds all clauses with that head and returns them as a set of clauses to be edited. While this option is only of limited use in Quintus Prolog (because the procedure being edited must be declared dynamic), in Symbolics Prolog it will allow you to edit any Prolog procedure.

Once you have called one of the three procedures that invoke the editor, you will enter the top level. This level is distinguished from lower levels in that you are not actually editing a Prolog term, but editing a set of terms. At this level you can perform operations on the set of clauses in the procedure like retracting an old clause, or asserting a new clause. [1]

---

[1] These changes to the database are not actually recorded until the editing session is finished.

The editor will report at every level what kind of structure you are editing. The kinds of structures that the editor knows about are:

A *Set* of clauses (top level only)
A *List* of terms
A *Conjunction* of terms (actually any infix right-associative operator)
A *Complex Term* (A functor followed by some number of arguments)
An *Atom*

It will then display the functor of the term (if appropriate) and the members of the term. Following are some examples:

Editing a Set of Rules
Rule1: objectbe::=(astg;nstg;pn),{sem_rep(append)}
Rule2: objectbe::=(vingo;venpass),{sem_rep(copy)};{dwh2},nullwh

Editing a term
Functor: ::=
Argument 1: objectbe
Argument 2: (vingo;venpass),{sem_rep(copy)};{dwh2},nullwh

Editing conjoined terms
Functor: ,
Term 1: {dwh2}
Term 2: nullwh

There are two types of commands that you can give to the structure editor: *Movement* commands and *Editing* commands. At every level in the editor, you are stationed at some Prolog term (except at the top level, when you are stationed at a set of terms). There are then two kinds of movement commands: downward movement and upward movement. A downward movement command is simply an integer that specifies which of the arguments of the current term you wish to move down to, from 1 to N (You can sometimes move to the 0th item, if the term has a functor then it is considered the 0th argument). At any term, only one direction is up, so the command 'u' will move you up one level in the structure. For convenience, the command 't' (for 'top') will move you to the top level.

An editing command is one which actually modifies the structure of the term that you are stationed at. The editing commands that are currently supported are:

delete
Specified by 'd<integer>'. This command deletes the named term.

insert-after
Specified by 'i<integer>'. This command inserts a new term after the mentioned term. 'i0' will make the new term the first argument. You will be prompted for the term that is to be inserted. As this new term is a Prolog term, you will have to end your input with a period.

replace
Specified by 'r<integer>'. This command replaces the specified term in the place of the mentioned term. *You can sometimes replace the 0th item,* if you want to change the functor of some complex term.

These commands are also available:

downward-movement
Specified by <integer>. Moves down to the *Nth* term of the current term.

move-up
Specified by 'u'. Moves up to the term that contains the current term.

go-to-top
Specified by 't'. Moves to the top level.

abort
Specified by 'a'. Ends the editing session and does not save any of the changes made!

print
Specified by 'p'. This command prints the structure of the current term. This command should only be used at the end of a command line.

help
Specified by '?'. This command prints out a listing of the available commands. It should only be used at the end of a command line.

If you command the editor to insert a whole rule, word, or clause, it will print out an entry from the set you are editing (to serve as a "template" of the type of entry you want to create) and ask you to edit that entry to form the new entry that you want to insert. (As a warning, do not insert a structure and then delete it. Instead, wait until the end of the editing session, and answer "no" to the query, "do you want to add ...?") If you try to replace a rule or clause, the editor moves to that rule or clause and asks you to replace each part individually (this is supposed to save you keystrokes).

To end the editing session, you must be at the top-most structure (a set of rules, a set of words, a set of clauses). At that level, type 'u' (or 't'), and the editor will ask you if you want to save the changes that you have made.

A SAMPLE EDITING SESSION:

```
| ?- edit_rule(objectbe).

Editing a Set of Rules
Rule1: objectbe::=(astg;nstg;pn),{sem_rep(append)}
Rule2: objectbe::=(vingo;venpass),{sem_rep(copy)};{dwh2},nullwh
Command: 2
                        /* edit the second rule

Editing a term
Functor: ::=
Argument 1: objectbe
Argument 2: (vingo;venpass),{sem_rep(copy)};{dwh2},nullwh
Command: 2
                        /* move to the second argument

Editing conjoined terms
Functor: ;
Term 1: (vingo;venpass),{sem_rep(copy)}
Term 2: {dwh2},nullwh
Command: 2 r2
                        /* replace the second argument of the second term

Replace the term: nullwh
with what Prolog term: stuff.

Editing conjoined terms
Functor: ,
Term 1: {dwh2}
Term 2: stuff
Command: t
                        /* go to the top level

Editing a Set of Rules
Rule1: objectbe::=(astg;nstg;pn),{sem_rep(append)}
Rule2: objectbe::=(vingo;venpass),{sem_rep(copy)};{dwh2},stuff
Command: t
                        /* go to the top again, i.e., finish editing this rule
                           ('t' and 'u' have the same effect at this level)

Do you want to replace: objectbe::=(vingo;venpass),{sem_rep(copy)};{dwh2},null
wh
with: objectbe::=(vingo;venpass),{sem_rep(copy)};{dwh2},stuff
Enter 'y' or 'n': n

If you have changed any grammar rules, you will have to either:
   1. Retranslate this rule.
   2. Switch the grammar to run interpreted only.
   3. Do nothing (and risk inconsistency!).
```

Please enter 1, 2, or 3: **3.**

yes

| ?- edit_word(replace).

Editing a set of words with the same root
Word 1: :(replace,root:replace,[v:[12],tv:[12,plural],12:[objlist:[nstgo,pn:[p
val:[with]],npn:[pval:[with]]]]])
Word 2: :(replaces,root:replace,[tv:[12,singular]])
Word 3: :(replaced,root:replace,[tv:[12,past],ven:[14],14:[12,pobjlist:[nullob
j,pn:[pval:[with]]]]])
Word 4: :(replacing,root:replace,[ving:[12]])
Command: **3**
/ edit the third word*

Editing a term
Functor: :
Argument 1: replaced
Argument 2: root:replace
Argument 3: [tv:[12,past],ven:[14],14:[12,pobjlist:[nullobj,pn:[pval:[with]]]]
]
Command: **3**
/ move to the third argument (a list)*

Editing a list
Element 1: tv:[12,past]
Element 2: ven:[14]
Element 3: 14:[12,pobjlist:[nullobj,pn:[pval:[with]]]]
Command: **i1**
/ insert an element into the list after the first element of the list*

What Prolog term should be inserted: **stuff.**

Editing a list
Element 1: tv:[12,past]
Element 2: **stuff**
Element 3: ven:[14]
Element 4: 14:[12,pobjlist:[nullobj,pn:[pval:[with]]]]
Command: **u**
/ go up one level*

Editing a term
Functor: :
Argument 1: replaced
Argument 2: root:replace
Argument 3: [tv:[12,past],**stuff**,ven:[14],14:[12,pobjlist:[nullobj,pn:[pval:[wi
th]]]]]
Command: **t**

```
Editing a set of words with the same root
Word 1: :(replace,root:replace,[v:[12],tv:[12,plural],12:[objlist:[nstgo,pn:[p
val:[with]],npn:[pval:[with]]]]])
Word 2: :(replaces,root:replace,[tv:[12,singular]])
Word 3: :(replaced,root:replace,[tv:[12,past],stuff,ven:[14],14:[12,pobjlist:[
nullobj,pn:[pval:[with]]]]])
Word 4: :(replacing,root:replace,[ving:[12]])
Command: u

Do you want to replace: :(replaced,root:replace,[tv:[12,past],ven:[14],14:[12,
pobjlist:[nullobj,pn:[pval:[with]]]]]).
with: :(replaced,root:replace,[tv:[12,past],stuff,ven:[14],14:[12,pobjlist:[nu
llobj,pn:[pval:[with]]]]]).
Enter 'y' or 'n': n

yes
```

```
| ?- edit_word(control).

Editing a set of words with the same root
Word 1: :(control,root:control,[n:[11,singular],v:[12],tv:[12,plural],11:[nonh
uman,h-change,h-norm],12:[objlist:[1],notnsubj:[2],vmanner,h-change,h-norm],1:
[nstgo,nsvingo,vingofn],2:[ntime1]])
Word 2: :(controlled,root:control,[tv:[12,past],ven:[14],14:[objlist:[1],notns
ubj:[2],vmanner,pobjlist:[3],h-change,h-norm],3:[nullobj]])
Word 3: :(controlling,root:control,[ving:[12]])
Word 4: :(controls,root:control,[n:[11,plural],tv:[12,singular]])
Command: i4
                        /* insert a word after the fourth word

Here is a word of the type that you want to create.
Edit it to make the new word.
Editing a term
Functor: :
Argument 1: control
Argument 2: root:control
Argument 3: [n:[11,singular],v:[12],tv:[12,plural],11:[nonhuman,h-change,h-nor
m],12:[objlist:[1],notnsubj:[2],vmanner,h-change,h-norm],1:[nstgo,nsvingo,ving
ofn],2:[ntime1]]
Command: r1
                        /* replace the first argument, in this case,
                           the word to be defined

Replace the term: control
with what Prolog term: controller.

Editing a term
Functor: :
Argument 1: controller
Argument 2: root:control
```

Argument 3: [n:[11,singular],v:[12],tv:[12,plural],11:[nonhuman,h-change,h-nor
m],12:[objlist:[1],notnsubj:[2],vmanner,h-change,h-norm],1:[nstgo,nsvingo,ving
ofn],2:[ntime1]]
Command: **r3**

/ replace the third argument, in this case, the definition list*

Replace the term: [n:[11,singular],v:[12],tv:[12,plural],11:[nonhuman,h-change
,h-norm],12:[objlist:[1],notnsubj:[2],vmanner,h-change,h-norm],1:[nstgo,nsving
o,vingofn],2:[ntime1]]
with what Prolog term: **[n:[11,singular],11:[human]].**

Editing a term
Functor: :
Argument 1: **controller**
Argument 2: root:control
Argument 3: **[n:[11,singular],11:[human]]**
Command: **t**

Editing a set of words with the same root
Word 1: :(control,root:control,[n:[11,singular],v:[12],tv:[12,plural],11:[nonh
uman,h-change,h-norm],12:[objlist:[1],notnsubj:[2],vmanner,h-change,h-norm],1:
[nstgo,nsvingo,vingofn],2:[ntime⊥]])
Word 2: :(controlled,root:control,[tv:[12,past],ven:[14],14:[objlist:[1],notns
ubj:[2],vmanner,pobjlist:[3],h-change,h-norm],3:[nullobj]])
Word 3: :(controlling,root:control,[ving:[12]])
Word 4: :(controls,root:control,[n:[11,plural],tv:[12,singular]])
Word 5: :(**controller**,root:control,[**n:[11,singular],11:[human]**])
Command: **t**

Do you want to add the word: :(controller,root:control,[n:[11,singular],11:[hu
man]]).
Enter 'y' or 'n': **y**

yes

System Administration for Pundit (SAP)

File: ~nlp/bin/SA_bin/README
Author: Korrinn Fu
Date: 3/23/89; 4/14/89; 4/17/89; 5/3/89

I. SAP Overview

This is our new tool for Pundit system administration. We named it "SAP", which stands for System Administration for Pundit. SAP is an interactive tool which will guide you through the system administration process. It provides menu choices for each step of the process, and you will no longer need to get printouts of other documentations in order to do system administration.

For those of you interested in seeing further documentation on the system administration process, look into the README files in ~nlp/NEWFILES/system_administration/command_files, and in its subdirectories NEWFILES_cf and pundit_cf. Another piece of useful information is in ~nlp/NEWFILES /system_administrator/checklist. Some of the information is outdated--we don't do system administration on the vax anymore. However, the checklist has an excellent overall description of the entire system administration process which SAP performs.

II. Software architecture

The software architecture of SAP is fully illustrated in a diagram--a gremlin figure depicting the flow of control of SAP is in SA_structure.grn. The shellscripts in ~nlp/bin/SA_bin are also fully documented with input/output parameters.

To look at this structure, all you have to do is (in suntools):

gremlin SA_structure.grn

or print it out on the imagen just like how you'd print other gremlin pictures. To print this file, all you have to do is:

1. create a file with the following 3 lines:
   .GS
   file SA_structure.grn
   .GE
2. print it out to image:
   grn <filename created in 1.> | ditroff -me -Pip<1 or 2>

III. How to use SAP

First, make sure you have access to the path ~nlp/bin (in your .login or .cshrc), or you would need to enter the entire path.

Second, you need to be user nlp to execute SAP. To do this, just type to unix prompt:

su nlp

and enter the password when prompted.

To start the system administration procedure up, type at the unix prompt:

sap

This will display an overall picture of the system administration process and a menu. The menu choices are:

1. make NEWFILES images
2. update pundit (update_pundit)
3. make stable images
4. clean up
5. undo and redo

Choice 1 allows you to make the NEWFILES images. Choice 2 activates the command update_pundit to move new files to the stable directory. (There is a man page for this command.) Choice 3 creates the stable pundit images. Choice 4 allows you to remove files that are no longer needed, and archive other files for future use. Choice 5 is a menu for NEWFILES and pundit, it allows you to do partial restart on images. You can choose to redo NEWFILES images, or pundit stable images; all the images of just a subset of the images.

Since each menu choice is rather self-explanatory, I'll not go into the details of each here. However, a brief description of each item is provided in section IV.

You are responsible for checking the image results after NEWFILES images, and after stable images are created. To do this, look at the ~nlp/NEWFILES/<domain>/<domain>_diff.test for NEWFILES, and into ~nlp/pundit/<domain>/<domain>_diff.test for the stable images. This provides information on any new change that the current image has, over the previous image; whether the current result is consistent with the result from the last administration. If there are differences, you'd need to check with the author(s) of the codes to see if the difference is intended.

IV. Files important to the administrator

SAP creates a number of log and err files during it's tour of system administration. The files of interested for an administrator to look at to monitor the progress is: (in ~nlp/bin/SA_bin)

NEWFILES:
- NEWFILES.log
- NEWFILES_time.log
- NEWFILES.err

pundit:
- pundit.log
- pundit_time.log
- pundit.err

redo:
- redo.log
- redo_time.log
- redo.err

The <NEWFILES, pundit, redo>.log files tells you which step SAP is at regarding the image making and testing process. These are the messages printed by SAP, at each different stage of making/testing an image.

The *time.log files tells you the time a process started/ended. The purpose of this file is to keep a time stamp on each step of making/testing an image. Before a process is started, the file gets

a time stamp. After a process is completed, the file gets another time stamp.

The *.err files are the diagnostic/error messages produced by SAP. By looking at the error messages of this file, you can tell whether there were any problems with making/testing the images, and what kind of problems they were.

V. Main menu choices

(1) SAP first checks if there is enough disk space, we have decided that 5% available disk space is required for us to have a successful image making round. If there isn't enough space, SAP would ask if you'd like to see a list of images under ~nlp, and even send a request to delete images out to the group if you'd like.

SAP allows a user to make all the images (pundit, casreps, muck, ships, trident, and opreps), or just a subset of the domain images. If only a subset of the images are created, a test_<domain>_<current date>.log file is created from the previous log. This is to ensure uniformity so the next time, SAP'd be able to find the proper log files.

(2) SAP provides update_pundit as a menu choice following 1. so you don't have to find out what comes after making NEWFILES images.

(3) After you have checked the NEWFILES images, you will go on to make the stable images. SAP allows a user to make all the images, or just a subset of domain images, just like that of NEWFILES.

(4) After you checked the stable images and they are correct, SAP cleans out the NEWFILES, stable and SAP's directory. This step will take more time as SAP prompts you for permission to "save" or "rm" before every erasure. This is the last step of the whole process.

(5) Each time after creating images (steps 1 and 3), you might find mistakes in the images, and you'd want to redo the incorrect images. SAP allows you to do partial restart via menu choices. It prompts you for (y/n) input, allowing you to redo all the images, or just a subset of the domain(s).

Control Flow

of sap